Linguistic side effects

A thesis presented by

Chung-chieh Shan

to the Division of Engineering and Applied Sciences
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the subject of computer science

Harvard University
Cambridge, Massachusetts
September 2005

Abstract

Apparently noncompositional phenomena in natural languages can be analyzed like computational side effects in programming languages: anaphora can be analyzed like state, intensionality can be analyzed like environment, quantification can be analyzed like delimited control, and so on. We thus term apparently noncompositional phenomena in natural languages *linguistic side effects*. We put this new, general analogy to work in linguistics as well as programming-language theory.

In linguistics, we turn the continuation semantics for delimited control into a new implementation of quantification in type-logical grammar. This graphically-motivated implementation does not move nearby constituents apart or distant constituents together. Just as delimited control encodes many computational side effects, quantification encodes many linguistic side effects, in particular anaphora, interrogation, and polarity sensitivity. Using the programming-language concepts of evaluation order and multistage programming, we unify four linguistic phenomena that had been dealt with only separately before: linear scope in quantification, crossover in anaphora, superiority in interrogation, and linear order in polarity sensitivity. This unified account is the first to predict a complex pattern of interaction between anaphora and raised-wh questions, without any stipulation on both. It also provides the first concrete processing explanation of linear order in polarity sensitivity.

In programming-language theory, we transfer a duality between expressions and contexts from our analysis of quantification to a new programming language with delimited control. This duality exchanges call-by-value evaluation with call-by-name evaluation, thus extending a known duality from undelimited to delimited control. The same duality also exchanges the familiar *let* construct with the less-familiar *shift* construct, so that the latter can be understood in terms of the former.

Contents

Abstract iii

List of Figures vi

Acknowledgments ix

Chapter 1. Introduction 1
1.1. Linguistics 1
1.2. Programming-language theory 2
1.3. Contributions 3
1.3.1. Evaluation order in natural languages 4
1.3.2. Delimited duality in programming languages 6
1.4. Environment in programming languages 7
1.5. Intensionality in natural languages 11
1.6. Computational versus linguistic side effects 15

Chapter 2. Formalities 19
2.1. Context-free grammars for natural language 21
2.2. A toy programming language 24
2.3. The λ-calculus 30
2.3.1. Confluence, normalization, and typing 35
2.3.2. The formulas-as-types correspondence 36
2.3.3. Products 38
2.3.4. Sums 40
2.3.5. Recursion 41
2.4. Type-logical grammar 44
2.4.1. Multimodal type-logical grammar 49

Chapter 3. The analogy: delimited control and quantification 55
3.1. Delimited control 55
3.1.1. A first attempt 57

3.1.2. Enforcing call-by-value, left-to-right evaluation 60
3.2. Continuations for delimited control 64
3.2.1. The continuation-passing-style transform 66
3.2.2. Types for continuations 71
3.2.3. Higher-order delimited control 76
3.3. Quantification 83
3.3.1. Generalized quantifiers 83
3.3.2. In-situ quantification 86
3.4. Delimited control versus quantification 91

Chapter 4. Evaluation order in natural languages 93
4.1. Type environments as graphs 94
4.2. Scopes apply inward; quantifiers apply outward 98
4.3. Refining the analysis of quantification by refining notions of context 103
4.3.1. Restrictors 104
4.3.2. Islands 106
4.4. Linear order and evaluation order 108
4.5. Beyond quantification 112
4.5.1. Anaphora and crossover 113
4.5.2. Questions and superiority 118
4.5.3. Polarity sensitivity 123
4.5.4. Reversing evaluation order 128

Chapter 5. Delimited duality in programming languages 129
5.1. Call-by-value versus call-by-name 130
5.2. Delimited control in a dual calculus 136
5.3. Restoring confluence 141
5.4. The continuation-passing-style transform 147

Chapter 6. Conclusion 149

Bibliography 151

List of Figures

1.1	Observations on a toy programming language for printing	3
1.2	A tiny programming language	7
1.3	A tiny programming language, with environment	9
1.4	A tiny fragment of English	11
1.5	A tiny fragment of English, with think	13
2.1	A natural-language fragment	22
2.2	The context-free grammar equivalent to Figure 2.1	22
2.3	A natural-language fragment that distinguishes among three kinds of verbs	23
2.4	The context-free grammar equivalent to Figure 2.3	23
2.5	A toy programming language for arithmetic	24
2.6	The context-free grammar equivalent to Figure 2.5	24
2.7	Expressions are trees but usually written with a textual representation	25
2.8	Proving that if true = succ succ 0 then false else 0 + 0 is well formed	26
2.9	Evaluation contexts	27
2.10	The context-free grammar of evaluation contexts, equivalent to Figure 2.9	27
2.11	The computation relation between expressions	27
2.12	Three confluent ways to run the same program	28
2.13	A toy programming language with a more refined type system	29
2.14	The untyped λ-calculus	31
2.15	Proving that $\lambda f.\, \lambda x.\, f(fx)$ is well formed	33
2.16	Evaluation contexts in the λ-calculus	33
2.17	The simply-typed λ-calculus	35
2.18	Proving that $\lambda f.\, \lambda x.\, f(fx)$ has type $(a \to a) \to (a \to a)$	36
2.19	Natural-deduction rules for propositional intuitionistic logic	37
2.20	Proving that $(a \to a) \to (a \to a)$	37

2.21	Proof reduction for $\rightarrow$I followed by $\rightarrow$E	38
2.22	Adding products to the λ-calculus	39
2.23	Adding unit to the λ-calculus	40
2.24	Adding sums to the λ-calculus	41
2.25	Natural-language derivations in the simply-typed λ-calculus, using structural rules to generate acceptable as well as unacceptable sentences	46
2.26	The non-associative Lambek calculus without products, with semantics	48
2.27	The non-associative Lambek calculus without products, without semantics	49
2.28	Adding a binary mode m to the Lambek calculus without products	51
2.29	Adding a unary mode m to the Lambek calculus	52
2.30	A multimodal type-logical grammar that generates the context-sensitive language of doubled strings $\{ ww \mid w \in \{\mathsf{a}, \mathsf{b}\}^* \}$	54
3.1	Defining evaluation subcontexts and redefining evaluation contexts	58
3.2	Defining values in the λ-calculus	60
3.3	Restricting the computation relation to enforce call-by-value, left-to-right evaluation in a λ-calculus with # and abort	63
3.4	The evaluation result $\lfloor V \rfloor$ of values V in the λ-calculus	64
3.5	The continuation-passing-style transform $[\![E]\!]$ for expressions E in the λ-calculus with # and abort, under call-by-value, left-to-right evaluation	67
3.6	Defining expressions in the λ-calculus with # and abort	72
3.7	Proving that $\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle$ is well formed	75
3.8	Proving that $\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle$ is well formed	75
3.9	Abstracting control	76
3.10	Alice thinks nobody saw Bob	85
3.11	Alice saw nobody's mother	87
3.12	Someone saw everyone	89
4.1	Adding the binary mode $\circledcirc$ to the Lambek calculus without products	95
4.2	Three ways to decompose $\Delta[\Theta, \Pi]$ into a subexpression and a context	98
4.3	Alice saw [ ]	99
4.4	Alice saw everyone	100
4.5	Alice saw everyone, without the formal bureaucracy	101

4.6 Linear scope for Someone saw everyone 101
4.7 Inverse scope for Someone saw everyone 102
4.8 Every school griped 105
4.9 Letting a school take narrow scope within dean of a school 106
4.10 Linear scope for Someone saw everyone, under left-to-right evaluation 110
4.11 Inverse scope for Someone saw everyone, under left-to-right evaluation, using unquotation 113
4.12 Everyone$_i$ saw his$_i$ mother 117
4.13 Deriving whose mother Alice saw from Alice saw whose mother 121
4.14 Who saw what 122
4.15 The negative clause (incomplete sentence) *np* saw anyone 126

5.1 An untyped λ-calculus with shift and reset, enforcing call-by-value, argument-to-function evaluation 131
5.2 An untyped λ-calculus with shift and reset, enforcing call-by-name, context-to-function evaluation 132
5.3 A dual calculus with delimited control operators 137
5.4 Translating the λ-calculus to the dual calculus 139
5.5 Duality in the dual calculus 141
5.6 A dual calculus with delimited control operators, enforcing call-by-value evaluation 142
5.7 A continuation-passing-style transform for the dual calculus 148

Acknowledgments

Chapter 4 is joint work with Chris Barker (Shan and Barker 2005; Barker and Shan 2005).

Incomparable thanks to Nate Ackerman, Amal Ahmed, David Ahn, Karlos Arregi, Peter Arvidson, Tamara Babaian, Chris Barker, Rachel Ben-Eliyahu-Zohary, Johan van Benthem, Raffaella Bernardi, William Byrd, Daniel Büring, Marco Carbone, Bob Carpenter, Balder ten Cate, Ruggiero Cavallo, Victoria Chang, David Chiang, Florin Constantin, Olivier Danvy, Veneeta Dayal, Elizabeth Denne, John Dias, Allyn Dimock, David Dowty, Matthias Felleisen, Andrzej Filinski, Kai von Fintel, Matthew Fluet, Danny Fox, Lisa Friedland, Daniel Friedman, Kobi Gal, Svetlana Godjevac, Rajeev Gore, Paul Govereau, Philippe de Groote, Barbara Grosz, Daniel Hardt, Robert Harper, Kelly Heffner, Irene Heim, Andre Henriques, David Hiniker, Glenn Holloway, C.-T. James Huang, Luke Hunsberger, Rebecca Hwa, Sabine Iatridou, Pauline Jacobson, Micha Josephy, Aravind Joshi, Gerhard Jäger, Emir Kapancı, Oleg Kiselyov, Samuel Klein, Shriram Krishnamurthi, Susumu Kuno, Sebastien Lahaie, François Lamarche, Jo-wang Lin, Shine Lin, Harry Mairson, Chris Manning, Jan Midtgaard, Michael Moortgat, Richard Moot, Greg Morrisett, Radhika Nagpal, Peter Neergaard, Rani Nelken, Rebecca Nesson, Brenda Ng, Lynn Nichols, Jill Nickerson, Andreea Nicoara, Jonathan Nissenbaum, Richard Oehrle, David Parkes, Francis Jeffry Pelletier, Fernando Pereira, David Pesetsky, Simon Peyton Jones, Avi Pfeffer, Andrew Pimlott, Chris Potts, Johnn Prance, Michael Rabin, Antonio Ramirez, Norman Ramsey, Tim Rauenbusch, Kevin Redwine, Seth Riney, Janet Rosenbaum, Wheeler Ruml, Amr Sabry, Ivan Sag, Margo Seltzer, Pao-ching Shan, Stuart Shieber, Olin Shivers, Jeffrey Siskind, Christian Skalka, Paul Steckler, Matthew Stone, Kaihsu Tai, Autrijus Tang, Xiaopeng Tao, Geetika Tewari, Hayo Thielecke, Chris Thorpe, Dylan Thurston, Jean-Baptiste Tristan, David Van Horn, Willemijn Vermaat, Philip Wadler, Mitchell Wand, Yanling Wang, Susan Wieczorek, Yoad Winter, and Noam Zeilberger.

This research is supported by the United States National Science Foundation Grants IRI-9712068 and BCS-0236592.

This version of the dissertation is typeset for use rather than archival. The two versions differ only in numbering and cross-references. All citations should be to this version.

CHAPTER 1

Introduction

This dissertation is about computational linguistics, in two senses. First, we apply insights from computer science, especially programming-language semantics, to the science of natural languages. Second, we apply insights from linguistics, especially natural-language semantics, to the engineering of programming languages. The phrase "computational linguistics" has a popular third sense, which is natural-language processing: teaching computers to listen to, speak, read, and write natural language. That is not our aim, even though the research described here indirectly helps it—by enhancing our understanding of natural and programming languages. This dissertation contains no quantitative performance measures. Rather, our success is measured by the qualitative yardsticks of linguistics and programming-language theory. Therefore, we introduce these yardsticks before describing our contributions in more detail.

1.1. Linguistics

Linguistics aims to explain empirical observations of natural language: what utterances are available for what meanings. For example, a linguist observing Harvard students may note that they can pronounce and perceive the utterances Alice passed and Bob passed without any sense of error, and that they react differently to the two utterances. But before analyzing or even reporting such observations, the linguist must focus on some aspects of language and abstract away from others. For example, Alice passed can be pronounced and perceived differently by different people in different situations with different shades of meaning. We pretend—as linguists often do—that there is a single language called English, spoken perfectly at an instant in time by a homogeneous community who pronounce Alice passed identically. We idealize—as natural-language syntacticians often do—the pronunciation of Alice passed as a two-word sequence, which speakers judge to be *acceptable* with little or no information about the situation of use. By contrast, we take the two-word sequence *passed Alice to be *unacceptable* (notated with the asterisk in front), in that it is only with a sense of error that an idealized community of English speakers can produce or comprehend the isolated utterance in an idealized situation of English speech.

As natural-language semanticists often do, we distinguish the meanings of

utterances by observing their *truth conditions*: when is an utterance true, and when is it false? That is, what situations (or models, or possible worlds) satisfy a given utterance's description? For example, we observe empirically that Alice passed and Bob passed differ semantically, because a situation exists where Alice passed is true but Bob passed is false. We also observe that Alice passed is true in every situation where Both Alice and Bob passed is true. (These are facts about English, in that only an English speaker knows them, but they presumably have to do with corresponding logical entailments and non-entailments.) Thus we aim to produce a scientific theory that is as simple as possible and accounts as accurately as possible for these observations of utterance acceptability and truth conditions. Like many linguists, we focus our theories on particular linguistic phenomena we are interested in and utterances that exemplify them, then expect to gain formal devices and informal insights that apply more broadly.

Our notions of *word* and *meaning* only intuitively resemble the colloquial senses of the terms, because we justify (operationalize) them only by how much they help us account for utterance acceptability and truth conditions. For example, we view Alice passed as a two-word sequence only because it makes for a simpler or more accurate account of utterance acceptability, not because standard English orthography puts a space between Alice and passed. Similarly, to determine the meaning of Alice, we weigh only what would make for a simpler or more accurate account of truth conditions, not our intuition that the utterance Alice passed is about the person Alice. Indeed, truth conditions only tell us that certain utterances have different meanings, not what these meanings are. Moreover, truth conditions fall silent on utterances that can be neither true nor false, such as Alice, Did Alice pass?, and If Alice passed. Yet as Section 1.5 illustrates, natural language is rich enough for us to find out about words and meanings just by striving to account for utterance acceptability and truth conditions simply and accurately. For example, the utterances mentioned above help us decide that Alice is a word and what it means.

1.2. Programming-language theory

Computer-science research on programming aims to help programmers convey their intentions, and computers execute them, correctly and efficiently. Programming-language theory contributes to this goal by modeling and reasoning about the conveyance and execution. As has proven useful, we abstract away from how programmers and computers interact and vary over time and space in different environments. Instead, we formalize a programming language as a collection of *well-formed* programs that *evaluate* to observable outcomes. Under this model, an idealized programmer conveys a well-formed program to an idealized computer, which runs the program and announces the outcome. Programs that are *ill-formed*

Well-formed program	Outcome	Ill-formed program	No outcome
2 + 3	5	2 +	×
print (2 + 3)	5, and printing 5	print	×
2 + (print 3)	5, and printing 3	2 + (3 print)	×

Figure 1.1. Observations on a toy programming language for printing

(that is, not well-formed) are prohibited.

As in linguistics, we focus our theories on issues and programs of interest, then expect to gain formal devices and informal insights that apply more broadly. For example, to study the issue of printing, we may contemplate a toy programming language with the programs and outcomes listed in Figure 1.1. An outcome in this language is a number and possibly some printed output.

Suppose that a demanding customer calls up a programming-language design shop and orders the observations given in Figure 1.1. The programming-language theorist at the shop then tries to design simple rules that give rise to the observations accurately, so as to teach programmers and build computers to convey and execute intentions in the language. These rules invoke notions of syntax and semantics to distinguish well-formed programs and determine their outcomes. For example, it can be useful to view parentheses (as in print (2 + 3)) as syntactic units ("words") in their own right, but only for some purposes (like checking for a well-formed program) and not others (like executing a well-formed program). Similarly, it can be useful to assign semantic values ("meanings") to program parts like 2, +, and print, but only to account for different outcomes (that is, to ensure that expressions with the same meaning have the same outcome). For example, the fact that the three well-formed programs in Figure 1.1 differ in outcome tells us only that they differ in meaning, not what they mean. Moreover, outcomes fall silent on program parts that are not complete programs, such as + and print. Yet as Section 1.4 illustrates, we can learn plenty about the syntax and semantics of a rich-enough programming language just by striving to specify its well-formed programs and their outcomes simply and accurately.

1.3. Contributions

This dissertation shows a new way for linguists and programming-language theorists to share their work and help each other: an analogy between *apparently noncompositional phenomena* in natural languages and *computational side effects* in programming languages. As is well-studied and explained below in this chapter, expressions like every student elude compositional treatment at first glance, and programs like print 3 incur computational side effects. It turns out that these

phenomena may be analyzed using similar tools. To stress this analogy, we term apparently noncompositional phenomena in natural languages *linguistic side effects*. We apply this analogy to linguistics as well as programming-language theory.

Many connections between natural and programming languages are based on logic, including this one. The reason is that logic underlies the syntax and semantics of both kinds of languages. On the programming-language side, programs can be viewed as logical proofs according to the *formulas-as-types correspondence* (Girard et al. 1989), as well as reasoned about using logical axioms (Hoare 1969). On the natural-language side, the proofs and models of logic can characterize the acceptability of utterances (Lambek 1958) as well as their truth conditions (Montague 1974a).

We introduce the logical machinery used in the bulk of this dissertation in Chapter 2. Then, in Chapter 3, we use this machinery to present the analogy between computational and linguistic side effects. In one direction, Chapter 4 draws from the execution of computer programs to model natural languages more realistically. In the opposite direction, Chapter 5 then draws from the symmetry of linguistic combinations to design programming languages more perspicuously. We evaluate our contributions and compare them with previous work throughout the dissertation, then collect the evaluations in Chapter 6, along with some thoughts for future work.

To be sure, this work is far from the first to draw an analogy between utterances and programs. *Intensional logic*, in which much natural-language semantics since Montague (1974a) is couched, has long been understood computationally (Hobbs and Rosenschein 1978; Hung and Zucker 1991). More recently, *dynamic semantics* (Groenendijk and Stokhof 1991; Heim 1982; Kamp 1981) has been used to analyze natural-language phenomena such as verb-phrase ellipsis (van Eijck and Francez 1995; Gardent 1991; Hardt 1999), as well as to design a programming language (van Eijck 1998). Our work is novel because the general concept of side effects unifies many concerns at both ends of the analogy.

The rest of this section gives more details on our contributions to the study of natural and programming languages.

1.3.1. Evaluation order in natural languages. Most computational side effects and many linguistic ones are thought of as the dynamic effect of executing a program or processing an utterance. This is the intuition underlying the term "side effects". For example, it is intuitive to conceive of *state* (a computational side effect) and *anaphora* (a linguistic side effect) in similar, dynamic terms, as follows. In the sentence

(1.1) Every woman's father saw her mother,

the pronoun her can refer to the woman introduced by the *antecedent* every woman. Thus (1.1) can mean that every woman x is such that x's father saw x's mother. A pronoun tends to appear only after its antecedent, so the sentence

(1.2) Her father saw every woman's mother

does not have the same meaning. To explain this prohibition against *crossover*, it is popular to hypothesize that every woman stores a *discourse referent* into a memory cell and her retrieves it, and that a human typically processes parts of an utterance in spoken order. This hypothesis is appealingly reminiscent of how the program

(1.3) $x := 2; x + 1$

stores a number into the variable x and later retrieves it, and how a computer typically executes parts of a program in written order. This view on anaphora and state is dynamic, as indicated by the preceding verbs "store", "retrieve", "process", and "execute". Unfortunately, the naïve form of this hypothesis fails to account for exceptions like the following.

(1.4) Which of her relatives did every woman see?

(1.5) Which woman did her father see?

In (1.4), the phrase every woman occurs after the pronoun her, yet can serve as its antecedent. In (1.5), the phrase which woman occurs before the pronoun her, yet cannot serve as its antecedent.

Relating this dynamic (*operational*) view on side effects to a static (*denotational*) view has been a focus of recent research. Across our new side-effects analogy, linguists and programming-language theorists have been asking alike: how does an expression manage to be both a static product that stands alone mathematically and a dynamic action that takes place physically (Wadler 1997; Trueswell and Tanenhaus 2005)? For programming languages, *game semantics* (Abramsky and McCusker 1997) and other models of interaction (Milner 1996) blur the boundary between what programs statically denote and how they dynamically operate. Linguists are constantly reminded that the distinction between semantics and pragmatics is far from black and white, by dynamic semantics and notions like the *felicity* of an answer to a question (Hamblin 1973; Groenendijk and Stokhof 1997) and the *computational load* to produce or comprehend an utterance (Gibson and Pearlmutter 1998; Altmann 1990).

In Chapter 4, we apply the dynamic concept of *evaluation order* from programming languages to natural languages. Using this concept, we properly formalize the hypothesis about anaphora above to account for crossover, including exceptions like (1.4) and (1.5). Moreover, the similarity between anaphora and state turns out to be just one in a wide range of similarities between linguistic

and computational side effects: *intensionality* can be analyzed like *environment*, *quantification* can be analyzed like *delimited control*, and so on. The single hypothesis that humans typically process utterance parts in spoken order, once we formalize it properly, turns out to explain an unprecedented variety of linguistic generalizations whose similarity across side effects was previously unrecognized: not just the prohibition against crossover in anaphora, but also the preference for *linear scope* in quantification, the *superiority* constraint on questions, and the effect of spoken order on *polarity sensitivity*. Via our side-effects analogy, programming-language theory helps us formalize this natural hypothesis regarding how humans process utterances dynamically.

1.3.2. Delimited duality in programming languages. Many linguistic formalisms feature a duality between left and right, that is, an involution that reverses the spoken order of utterance parts. *Type-logical grammar*, the linguistic formalism used in this dissertation, is one such formalism whose left-right duality is especially obvious. For example, given a model of English in type-logical grammar, it is trivial to mirror it to yield a model of a hypothetical natural language that is just like English, except with the opposite word order, so *Alice passed is not an acceptable utterance, but passed Alice is.

The left-right duality in type-logical grammar is intrinsic to the *default mode* of binary combination: juxtapose two utterances to form a larger one. To analyze the linguistic side effect of quantification, in Chapter 4 we add another mode of binary combination: plug a subexpression into a context to form a larger expression. This *multimodal* type-logical grammar thus features another intrinsic duality, between the subexpression inside a context and the context outside a subexpression.

In Chapter 5, we apply the duality between inside and outside from natural languages to programming languages. Drawing from our linguistic analysis of quantification, we formalize a programming language with the computational side effect of delimited control, in which subexpressions and contexts are dual. It turns out that this duality exchanges *call-by-value* and *call-by-name*, two evaluation orders long studied in programming languages, so that one order may be understood in terms of the other. This result is not surprising, because call-by-value and call-by-name are known to be dual in the presence of *undelimited* control (Filinski 1989a,b; Danos et al. 1995; Curien and Herbelin 2000; Selinger 2001; Wadler 2003). Via our side-effects analogy, linguistics helps us extend this duality to delimited control for the first time. The same duality also exchanges the well-understood *let* construct for variable binding and the less-well-understood *shift* construct for delimited control, so that one construct may be understood in terms of the other.

The rest of this chapter informally illustrates our analogy between compu-

Well-formed program	Outcome
2 > 6	false
2 + 6 > 6	true
2 + 6 > (2 + 2) + 2	true
2 + 6 > 6 + 6	false
2 > 2	false

Figure 1.2. A tiny programming language

tational and linguistic side effects. We examine how a denotational semantics typically treats *environment* (a computational side effect) in Section 1.4, and *intensionality* (a linguistic side effect) in Section 1.5. We then point out how the treatments are similar in Section 1.6.

1.4. Environment in programming languages

Figure 1.2 illustrates a tiny programming language where arithmetic expressions are compared to yield Boolean results (true or false). For example, the program

(1.6) 2 + 6 > 6
'(Check if) the sum of 2 and 6 is greater than 6.'

evaluates to the true outcome.

Although arithmetic expressions like 2 + 6 and 6 are not complete programs with (Boolean) outcomes, it is intuitive to regard them as evaluating to numeric results. For example, we regard 2 + 6 as evaluating to 8, and 6 as evaluating to 6, even though these results are only indirectly observable by comparison using >. Under this intuition, if two expressions E_1 and E_2 evaluate to the same result in this language, then whenever E_1 occurs as part of a larger expression, it can be replaced with E_2 without affecting what the larger expression evaluates to.[1] For example, since 6 and (2 + 2) + 2 evaluate to the same result, so do 2 + 6 > 6 and 2 + 6 > (2 + 2) + 2.

In operational terms, each expression is a procedure that can be followed to produce a result. In denotational terms, we can take each expression simply to mean this result. For example, the program (1.6) can simply denote true. To provide a denotational semantics for a language is to specify the denotation $[\![E]\!]$ of every expression E. For our programming language, we specify the integer

[1] This property is sometimes known as *referential transparency* (Quine 1960), but we avoid the term because it is problematically overloaded (Søndergaard and Sestoft 1990, 1992).

and Boolean denotations

$$[\![\mathsf{2}]\!] = 2, \tag{1.7}$$

$$[\![\mathsf{6}]\!] = 6, \tag{1.8}$$

$$[\![A + B]\!] = [\![A]\!] + [\![B]\!], \tag{1.9}$$

$$[\![A > B]\!] = \begin{cases} \text{true} & \text{if } [\![A]\!] > [\![B]\!], \\ \text{false} & \text{if } [\![A]\!] \leq [\![B]\!], \end{cases} \tag{1.10}$$

where A and B are any expressions. These rules together entail that the program (1.6) denotes true.

$$[\![\mathsf{2} + \mathsf{6}]\!] = [\![\mathsf{2}]\!] + [\![\mathsf{6}]\!] = 2 + 6 = 8. \tag{1.11}$$

$$\begin{aligned} [\![\mathsf{2} + \mathsf{6} > \mathsf{6}]\!] &= \begin{cases} \text{true} & \text{if } [\![\mathsf{2} + \mathsf{6}]\!] > [\![\mathsf{6}]\!] \\ \text{false} & \text{if } [\![\mathsf{2} + \mathsf{6}]\!] \leq [\![\mathsf{6}]\!] \end{cases} \\ &= \text{true, because } 8 > 6. \end{aligned} \tag{1.12}$$

We want a denotational semantics that is *sound* with respect to the operational semantics, and this one is. Soundness means that, if two expressions E_1 and E_2 have the same denotation, then they evaluate to the same result. For example, since $\mathsf{2} > \mathsf{6}$ and $\mathsf{2} + \mathsf{6} > \mathsf{6} + \mathsf{6}$ both denote false, soundness guarantees (correctly) that they evaluate to the same result (namely false). Soundness is desirable because expressions with the same denotation should be equivalent in meaning, and what an expression evaluates to (in particular, the outcome of a program) should be part of its meaning.

Another desirable feature of this denotational semantics is that it is *compositional*: the denotation of every complex expression is fully determined by the denotations of its parts and the mode of combination. For example, the denotation of every + expression is fully determined by the denotations of its two subexpressions. Compositional semantic theories are desirable because they are easier to use and extend (Janssen 1997). In theory, any language admits a compositional semantics, regardless of what its expressions are and what they evaluate to—just let each expression denote itself.[2] However, it is often a challenge to come up with a compositional semantics that is simple and insightful enough to facilitate further analysis of the language.

[2] A less trivial result, shown by Zadrozny (1994) and Lappin and Zadrozny (2000) using non-well-founded set theory, is that any language admits a compositional semantics that *respects synonymy*. For a denotational semantics to respect synonymy is for two expressions E_1 and E_2 to denote the same thing whenever they are *synonymous* (or *observationally equivalent*). We say that E_1 and E_2 are synonymous if, whenever E_1 occurs as part of a larger expression, it can be replaced with E_2 to form another expression with the same evaluation result, and vice versa.

Well-formed program	Outcome
2 > 6	false
2 + 6 > 6	true
2 + 6 > (2 + 2) + 2	true
2 + 6 > 6 + 6	false
2 > 2	false
let it be 2. (it > 6)	false
let it be 2. (it + 6 > 6)	true
let it be 2. (it + 6 > (let it be it + it. it + 2))	true
let it be 6. (2 + it > it + it)	false
it + 6 > 6	false
it > 2	false
let it be 6. (it > 2)	true
let it be 6. (2 > 2)	false

Figure 1.3. A tiny programming language, with environment

We now add the *environment* feature to the language, as illustrated in Figure 1.3. We introduce two new constructs: a primitive expression it, which reads a number from the environment, and a binary operator let it be *A*. *B*, which provides *A* as the environment to be read by *B*. The program

(1.13) let it be 2. (it + 6 > 6)
'As for 2, the sum of it and 6 is greater than 6.'

evaluates to the true outcome, as 2 + 6 > 6 does, because let it be 2 provides the environment 2 to the expression

(1.14) it + 6 > 6
'The sum of it and 6 is greater than 6.'

As another example, the program

(1.15) let it be 2. (it + 6 > (let it be it + it. it + 2))

evaluates to the true outcome, as 2 + 6 > (2 + 2) + 2 does, because let it be it + it provides the environment 4 to the expression it + 2. We stipulate that any it not enclosed by let it be ... evaluates to 0. For example, the program (1.14) above evaluates to false, because 0 + 6 is not greater than 6. (On the other hand, a let it be ... that encloses no it is simply ignored.)

In denotational terms, once environment is introduced into the language, we can no longer just let each expression denote its result—at least not if we want the denotational semantics to stay compositional and sound, which we do. To see this, suppose that each program were to denote its Boolean outcome. Then, since the two programs in (1.16) both evaluate to false, they have the same denotation. By compositionality, the two programs in (1.17) must also have the same denotation. But (1.17a) evaluates to true, whereas (1.17b) evaluates to false, so soundness fails.

(1.16) a. it > 2
'It is greater than 2.'
b. 2 > 2
'2 is greater than 2.'

(1.17) a. let it be 6. (it > 2)
'As for 6, it is greater than 2.'
b. let it be 6. (2 > 2)
'As for 6, 2 is greater than 2.'

It is crucial to this proof that, whereas the programs in (1.16) evaluate to the same result, the programs in (1.17) do not. That is, the proof relies on two expressions E_1 and E_2 that evaluate to the same result, such that replacing E_1 with E_2 as part of a larger expression affects what the larger expression evaluates to. Adding environment made it impossible to specify a compositional and sound denotational semantics where each expression denotes its result. A *computational side effect* is a programming-language feature that gives rise to two expressions that evaluate to the same result by themselves but yield different results as part of a larger expression, thus rendering unsound any compositional denotational semantics where each expression denotes its result. In this sense, environment is a computational side effect.

To preserve soundness and compositionality in the presence of environment, the programming-language semanticist typically revises the denotational semantics so that an expression denotes not a result but a map from environments to results. For example, the expression 2 will now denote not 2 but the function that maps every environment to 2. Formally, we specify (cf. (1.7)–(1.10))

$$[\![2]\!](\rho) = 2, \tag{1.18}$$

$$[\![6]\!](\rho) = 6, \tag{1.19}$$

$$[\![A + B]\!](\rho) = [\![A]\!](\rho) + [\![B]\!](\rho), \tag{1.20}$$

$$[\![A > B]\!](\rho) = \begin{cases} \text{true} & \text{if } [\![A]\!](\rho) > [\![B]\!](\rho), \\ \text{false} & \text{if } [\![A]\!](\rho) \leq [\![B]\!](\rho), \end{cases} \tag{1.21}$$

Acceptable utterance	Truth value
Alice saw Bob	true
Alice saw the morning star	false
Alice saw the evening star	false
Bob saw Alice	true
Bob saw the morning star	true
Bob saw the evening star	true

Figure 1.4. A tiny fragment of English

for any number ρ and any expressions A and B. Having lifted denotations from plain results to environment-to-result functions, we can now provide denotations for expressions involving it:

$$[\![\text{let it be } A.\ B]\!]\,(\rho) = [\![B]\!]\,([\![A]\!]\,(\rho)), \tag{1.22}$$

$$[\![\text{it}]\!]\,(\rho) = \rho. \tag{1.23}$$

These rules give the desired denotation for the program (1.13).

$$[\![\text{it} + 6]\!]\,(2) = [\![\text{it}]\!]\,(2) + [\![6]\!]\,(2) = 2 + 6 = 8. \tag{1.24}$$

$$[\![\text{it} + 6 > 6]\!]\,(2) = \begin{cases} \text{true} & \text{if } [\![\text{it} + 6]\!]\,(2) > [\![6]\!]\,(2) \\ \text{false} & \text{if } [\![\text{it} + 6]\!]\,(2) \leq [\![6]\!]\,(2) \end{cases} \tag{1.25}$$
$$= \text{true, because } 8 > 6.$$

$$[\![\text{let it be 2. (it} + 6 > 6)]\!]\,(\rho) = [\![\text{it} + 6 > 6]\!]\,([\![2]\!]\,(\rho)) \tag{1.26}$$
$$= [\![\text{it} + 6 > 6]\!]\,(2) = \text{true}.$$

We have seen that adding environment to a programming language forces us to revise our denotational semantics, and how to model environment using functions as denotations. We have also seen that, even when a toy programming language is so simple that the outcome of a program can only be true or false, we can learn about the syntax and semantics of the language by striving to specify the programs and their outcomes simply and accurately. In the next section, we turn to natural language and learn about syntax and semantics by striving to specify which sentences are true and which are false simply and accurately.

1.5. Intensionality in natural languages

Figure 1.4 shows a tiny fragment of English, comprised of utterances about seeing (Frege 1891, 1892; Quine 1960). According to the table, Alice and Bob

saw each other, but only Bob saw Venus. (Venus is also known as the morning star and the evening star.)

When an utterance is true or false, like those in Figure 1.4, we say that the utterance *refers* to its truth value. Although expressions like Alice and the morning star are not utterances with (Boolean) truth values, it is intuitive to regard them as referring to objects. For example, we regard Alice as referring to Alice, and the morning star as referring to Venus, even though these references are only indirectly observable by embedding these expressions in a larger utterance with a truth value. Under this intuition, if two expressions E_1 and E_2 in this fragment of English refer to the same thing, then whenever E_1 occurs as part of a larger expression, it can be replaced with E_2 without affecting what the larger expression refers to. For example, since the morning star and the evening star have the same reference, so do Bob saw the morning star and Bob saw the evening star.

In operational terms, an expression like Alice and the morning star is a procedure that can be followed to pick out an object, and an expression like

(1.27) Alice saw the morning star

(1.28) Alice saw the evening star

is a procedure that can be followed to check if the expression is true. The sentence (1.27) means to pick out Alice and the morning star, then check whether she saw it. In denotational terms, we can take each expression simply to mean its reference. For example, the utterance (1.27) can simply denote false. We specify

$$[\![\text{Alice}]\!] = \text{Alice}, \tag{1.29}$$

$$[\![\text{Bob}]\!] = \text{Bob}, \tag{1.30}$$

$$[\![\text{the morning star}]\!] = \text{Venus}, \tag{1.31}$$

$$[\![\text{the evening star}]\!] = \text{Venus}, \tag{1.32}$$

$$[\![A \text{ saw } B]\!] = \begin{cases} \text{true} & \text{if } [\![A]\!] = \text{Alice and } [\![B]\!] = \text{Bob}, \\ \text{false} & \text{if } [\![A]\!] = \text{Alice and } [\![B]\!] = \text{Venus}, \\ \text{true} & \text{if } [\![A]\!] = \text{Bob and } [\![B]\!] = \text{Alice}, \\ \text{true} & \text{if } [\![A]\!] = \text{Bob and } [\![B]\!] = \text{Venus}, \ldots. \end{cases} \tag{1.33}$$

These rules together entail that the utterance (1.27) denotes false.

This denotational semantics is sound. For example, since the morning star and the evening star denote the same thing, they also refer to the same thing: the morning star is the evening star. This semantics is also compositional, as is evident from the form of (1.29)–(1.33).

We now expand this fragment of English to include some utterances about thinking, as shown in Figure 1.5. According to the table, Bob (being ignorant of

Acceptable utterance	Truth value
Alice saw Bob	true
Alice saw the morning star	false
Alice saw the evening star	false
Bob saw Alice	true
Bob saw the morning star	true
Bob saw the evening star	true
Bob thinks Alice saw the morning star	true
Bob thinks Alice saw the evening star	false

Figure 1.5. A tiny fragment of English, with think

astronomy) does not know that the morning star is the evening star, so he thinks Alice saw the morning star but does not think that Alice saw the evening star.[3]

In denotational terms, once we expand our fragment to include think, we can no longer just let each expression denote its reference—at least not if we want the denotational semantics to stay compositional and sound, which we do. To see this, suppose that each utterance that is true or false were to denote its Boolean reference. Then, since the utterances (1.27) and (1.28) have the same reference (namely false), they have the same denotation. By compositionality, the two utterances

(1.34) Bob thinks Alice saw the morning star

(1.35) Bob thinks Alice saw the evening star

must also have the same denotation. But (1.34) is true, yet (1.35) is false, so soundness fails.

It is crucial to this proof that, whereas the utterances (1.27) and (1.28) have the same reference, the utterances (1.34) and (1.35) do not. That is, the proof relies on two expressions E_1 and E_2 that have the same reference, such that replacing E_1 with E_2 as part of a larger expression affects the larger expression's reference. In English, think makes it impossible to specify a compositional and sound denotational semantics where each expression denotes its reference. The word think is said to be *intensional*.

To preserve soundness and compositionality in the presence of intensionality, the natural-language semanticist typically revises the denotational semantics so

[3]This classical example may be counter-intuitive to a reader who knows astronomy. An equivalent example is Bob thinks Alice drove on Route 9 versus Bob thinks Alice drove on Worcester Street.

that an expression denotes not a reference but a map from *possible worlds* to references. The morning star and the evening star will now denote not Venus but two distinct functions from possible worlds to heavenly bodies. These two denotations will map the actual world to the same heavenly body Venus, but other worlds to different heavenly bodies. Similarly, (1.27) and (1.28) will now denote not false but two distinct functions from possible worlds to truth values. These two denotations will map the actual world to false, but other worlds to different truth values. If w is a world where the morning and evening stars are distinct and Alice only saw the former, then the denotations of (1.27) and (1.28) map w to true and false, respectively. Formally, we specify (cf. (1.29)–(1.33))

(1.36) $[\![\text{Alice}]\!](w) = \text{Alice},$

(1.37) $[\![\text{Bob}]\!](w) = \text{Bob},$

(1.38) $[\![\text{the morning star}]\!](w) = \text{the morning star in the world } w,$

(1.39) $[\![\text{the evening star}]\!](w) = \text{the evening star in the world } w,$

(1.40) $[\![A \text{ saw } B]\!](w) = \text{whether } [\![A]\!](w) \text{ saw } [\![B]\!](w).$

Having lifted denotations from plain references to possible-world-to-reference functions, we can now provide denotations for utterances involving think:

(1.41) $[\![A \text{ think } B]\!](w) = \text{whether } [\![B]\!](w') \text{ is true for every world } w'$ that $[\![A]\!](w)$ considers possible in w.

These rules give the desired denotation for the sentence (1.34).

(1.42) $[\![\text{Bob thinks Alice saw the morning star}]\!](w)$

= whether $[\![\text{Alice saw the morning star}]\!](w')$ is true for every world w' that $[\![\text{Bob}]\!](w)$ considers possible in w

= whether $[\![\text{Alice}]\!](w')$ saw $[\![\text{the morning star}]\!](w')$ for every world w' that Bob considers possible in w

= whether Alice saw the morning star in every world that Bob considers possible in w

We have seen that the presence of an intensional verb in natural language forces us to revise our denotational semantics, and how to model intensionality using functions as denotations. We have also seen that we can learn about the syntax and semantics of natural language just by striving to specify the sentences and their truth conditions simply and accurately. We now compare the situation with that for programming languages.

1.6. Computational versus linguistic side effects

Environment constructs in programming languages (Section 1.4) and intensional verbs in natural languages (Section 1.5) exemplify a general pattern. In natural-language semantics, it is intuitive to expect every expression to have a *reference*, like a physical object or a Boolean value. In programming-language semantics, it is intuitive to model program evaluation by assigning to every expression a *result*, like a number or a Boolean value. (We henceforth use the terms "reference" and "result" interchangeably.) Given a notion of reference, some simple languages guarantee that, if two expressions E_1 and E_2 have the same reference, then whenever E_1 occurs as part of a larger expression, it can be replaced with E_2 without affecting the reference of the larger expression.

A denotational semantics is a system that assigns a denotation to every expression. We want such a semantics to be *sound*: two expressions with the same denotation should also have the same reference. We also want it to be *compositional*: the denotation of every complex expression should be determined by the denotations of its parts and the mode of combination. We prefer for every expression simply to denote its reference, but to preserve soundness and compositionality in the absence of the guarantee above requires that denotations be more complex than simple reference. Indeed, some expressions, such as Sherlock Holmes and the king of France, seem to have no reference at all, let alone a reference to denote soundly and compositionally. When such expressions are present in a language, it is obviously impossible for every expression to denote its reference.

Intensionality is only one of many natural-language phenomena that break the notion of reference or the guarantee above. We term them *linguistic side effects*. Some examples are *anaphora* (1.43), *quantification* (1.44), *interrogation* (1.45), *focus* (1.46), and *presuppositions* (1.47).

(1.43) Anaphora: A man walks in the park. He whistles.

(1.44) Quantification: Every woman whistles.

(1.45) Interrogation: Which star did Alice see?

(1.46) Focus: Alice only saw *Venus*.

(1.47) Presuppositions: The king of France whistles.

To account for each linguistic side effect, semantic theories typically preserve soundness and compositionality by incorporating into denotations some new aspect of meaning beyond reference.

The same guarantee discussed above is not just *observed* to break down in natural languages, but also *designed* to break down in programming languages: to make programs more concise and modular, we often want some expressions

to evaluate to the same result or no result at all, yet, when placed inside some larger context, to yield larger expressions that evaluate to different results. Environment is only one of many programming-language features that break the notion of evaluation result or the guarantee above. Such features are known technically as *computational side effects*, and the vast majority of programming languages in use today have them. Some examples are *output* (1.48), *state* (1.49), *nondeterminism* (1.50), *exceptions* (1.51), and *control* (1.52).

(1.48) Output: print 2; 10
'Print the number 2, then produce the number 10.'

(1.49) State: x := 2; 10
'Store 2 in the variable x, then produce 10.'

(1.50) Nondeterminism: 2 + random(10, 20)
'Add 2 to either 10 or 20, randomly chosen.'

(1.51) Exceptions: try(2 + throw) catch 3
'Add 2 to an error; fall back to 3 in case of error.'

(1.52) Control: label: 2 + goto label
'Add 2 to the result of starting over again.'

To preserve soundness and compositionality in the face of a computational side effect, semantic theories of programming languages typically take the same strategy as those of natural languages: they add to denotations a new aspect of meaning beyond evaluation results.

As described above, environment and intensionality can be modeled using functions as denotations. Other side effects call for different treatments. The proper treatments of many linguistic and computational side effects are open areas for research: some studies focus on a single side effect, while others characterize more than one.

Be it for the scientific goal of characterizing humans who learn and use language or for the engineering goal of building machines that process and execute language, the linguist and the programming-language theorist both strive for a simple, modular, and extensible treatment of side effects. One approach towards such a treatment is to unify multiple notions or phenomena and show how they are instances of a more general concern. That is the approach we take throughout this dissertation, both in drawing a general analogy between computational and linguistic side effects and in applying the analogy to its two ends:

- We use apparent noncompositionality to unify computational and linguistic side effects.
- We use left-to-right evaluation in the presence of computational side

effects to unify four generalizations across linguistic side effects: the prohibition against crossover in anaphora, the preference for linear scope in quantification, the superiority constraint on interrogation, and the effect of spoken order on polarity sensitivity.

- We use the duality between inside and outside in quantification (a linguistic side effect) to unify call-by-value with call-by-name, and the *let* construct with the *shift* construct, in the presence of delimited control (a computational side effect).

But first, we need some notation.

CHAPTER 2

Formalities

This chapter establishes the formal frameworks in which the remainder of this dissertation presents analyses of programming and natural languages. An *analysis* is comprised of general rules that specify or predict what expressions are acceptable and what they mean. Because these rules need to apply to a potentially infinite number of expressions, we cannot simply enumerate every case. Instead, we specify a finite set of inference rules, which can give rise to an infinite number of conclusions.

As is standard in logic, we indicate inference by a horizontal rule that separates zero or more premises above from exactly one conclusion below. Whereas the inference

$$\frac{\text{All men are mortal} \quad \text{Socrates is a man}}{\text{Socrates is mortal}} \tag{2.1}$$

concludes with mortality, the inference

$$\frac{\textsf{Alice} \text{ is a subject} \quad \textsf{left} \text{ is a predicate}}{\textsf{Alice left} \text{ is a sentence}} \tag{2.2}$$

concludes with acceptability. These premises and conclusions are called *judgments*.

This notation can be used to write inference rules as well as proofs. An *inference rule* describes an allowed pattern of inference. For example, the inference (2.1) illustrates the rule

$$\frac{\text{All men are } P \quad X \text{ is a man}}{X \text{ is } P}\ \text{All-Men}, \tag{2.3}$$

where P and X are called *metavariables* because they range over expressions that form a judgment (such as "mortal" and "Socrates"), not the objects they refer to (such as the set of mortals or the person Socrates). The optional label "All-Men" names the rule. We say that the inference (2.1) *instantiates* (or *matches*) the All-Men rule. To take another example, the inference (2.2) illustrates the rule

$$\frac{S \text{ is a subject} \quad P \text{ is a predicate}}{S\ P \text{ is a sentence}}\ \text{Subject-Predicate}, \tag{2.4}$$

where $S\ P$ means to concatenate S and P. A rule with no premise is an *axiom*. To reason about English, we might stipulate the axiom

$$\frac{}{\textsf{Alice is a subject}}\ \text{Alice.} \tag{2.5}$$

An axiom like this, which classifies an atomic expression like Alice, is called a *lexical entry* because it can be thought of as an entry in a lexicon (that is, dictionary).

A *proof* is a tree of connected inferences; it is also sometimes called a *derivation*. For example, the Subject-Predicate rule (2.4) and the Alice rule (2.5) can be instantiated and combined to form the proof

$$\frac{\dfrac{}{\textsf{Alice is a subject}}\ \text{Alice} \qquad \textsf{thinks vanilla is a predicate}}{\textsf{Alice thinks vanilla is a sentence}}\ \text{Subject-Predicate.} \tag{2.6}$$

Starting from the sole premise judgment that "thinks vanilla is a predicate", this proof concludes that "Alice thinks vanilla is a sentence". That "Alice is a subject" is not a premise assumed by the proof, but the conclusion of the Alice rule used by the proof. The proof works whether or not thinks vanilla is a predicate: it only says that Alice thinks vanilla is a sentence *if* thinks vanilla is a predicate.[1]

Our formal plan, then, is as follows. To specify a language, we introduce some judgment forms and enumerate some inference rules whose premise and conclusion patterns are of these forms. A typical judgment classifies an expression into a *type* (or *category*). For example, the judgment "Alice is a subject" above classifies the expression Alice into the type of subjects. We classify expressions into types so as to reason about similar expressions, such as all subjects or all predicates, together.

The inference rules we enumerate can be instantiated and combined to form proofs that (starting from no premise) classify certain expressions into the type of complete programs or complete utterances. We take exactly these expressions to be complete programs in our programming language or complete utterances in our natural language. Further, we describe what these programs mean, often by specifying how they execute, or what these utterances mean, often by modeling when they are true. A *grammar* is a formal system of inference rules and meaning descriptions.

When a given grammar classifies an expression into a type, we say that the expression is *well typed*. Contrary to some usage in computer science and linguistics, this classification is a syntactic discipline (Reynolds 1983; Pierce 2002) rather than a semantic one, in that it is an expression that is classified rather than its meaning. For example, the judgment "Alice is a subject" classifies Alice rather than Alice. Thus we use the terms "well-typed" and "well-formed"

[1] In response to What is Bob's favorite ice cream flavor?, for instance.

interchangeably. *Ill-typed* (or *ill-formed*) expressions are those that are not classified into any type. A grammar models them simply by their absence.

Like all models, a natural-language grammar only approximates reality: it concentrates on aspects of natural language that we want to study, leaving out the rest. In particular, in this dissertation we examine how words combine with each other (roughly speaking) without worrying about how they sound, originate, change, or vary from speaker to speaker. Hence we do not try to enumerate or derive every lexical entry like the Alice rule in (2.5). Instead, in most natural-language derivations below, we simply start from premises like "Alice is a subject", rather than starting from no premise or drilling down into the internal constitution of the word Alice. This approximation simplifies our grammar while still letting us study how words combine with each other. Similarly, a programming-language grammar also only approximates reality. Even if we wanted to, we could not perfectly formalize practical programming with all the features and complications that it entails, such as interactions with users, cosmic rays, and the evolution of a programming language over time. In order to make progress, it is standard practice to model programming languages with *toy* programming languages and natural languages with natural-language *fragments*. In either case, we design and study formal grammars that bring into focus the phenomena of interest in real-world languages.

The main body of this dissertation builds on the *λ-calculus* as a programming language and uses *type-logical grammar* to analyze natural languages. Before describing the λ-calculus (in Section 2.3) and type-logical grammar (in Section 2.4), we first introduce some concepts through a context-free fragment of natural language (in Section 2.1) and a toy programming language for arithmetic (in Section 2.2). Although we specify these systems completely, the reader who has not encountered inference rules and the λ-calculus before may find it helpful to acquire more background and intuition from an introductory text like Wadler's article (2000), Gallier's tutorial (1993, 1991), and Carpenter's (1997), Girard et al.'s (1989), and Pierce's (2002) books.

2.1. Context-free grammars for natural language

Figure 2.1 shows some inference rules that model a tiny fragment of English. There is only one judgment form, written $A : T$ and pronounced "A is a T", where A is an expression and T is a type. For now, an expression is a sequence of words, such as the empirically acceptable Alice said Bob left and the empirically unacceptable *Alice said Bob said. As mentioned earlier, types classify expressions. The types in this system are named after classes of expressions in linguistics: *np* (noun phrase), *vp* (verb phrase), *v* (verb), and *s* (clause, or sentence). For example,

Expressions $A : T$

$$\frac{}{\mathsf{Alice} : np} \quad \frac{}{\mathsf{Bob} : np} \quad \frac{}{\mathsf{left} : v} \quad \frac{}{\mathsf{slept} : v} \quad \frac{}{\mathsf{saw} : v} \quad \frac{}{\mathsf{loved} : v} \quad \frac{}{\mathsf{said} : v}$$

$$\frac{A : np \quad B : vp}{A\ B : s} \qquad \frac{A : v}{A : vp} \qquad \frac{A : v \quad B : np}{A\ B : vp} \qquad \frac{A : v \quad B : s}{A\ B : vp}$$

Figure 2.1. A natural-language fragment

$$np ::= \mathsf{Alice} \mid \mathsf{Bob} \qquad v ::= \mathsf{left} \mid \mathsf{slept} \mid \mathsf{saw} \mid \mathsf{loved} \mid \mathsf{said}$$
$$s ::= np\ vp \qquad vp ::= v \mid v\ np \mid v\ s$$

Figure 2.2. The context-free grammar equivalent to Figure 2.1

the judgment "Alice said Bob left : s" has the proof below.

(2.7)
$$\dfrac{\dfrac{}{\mathsf{Alice} : np} \qquad \dfrac{\dfrac{}{\mathsf{said} : v} \qquad \dfrac{\dfrac{}{\mathsf{Bob} : np} \quad \dfrac{}{\mathsf{left} : vp}}{\mathsf{Bob\ left} : s}}{\mathsf{said\ Bob\ left} : vp}}{\mathsf{Alice\ said\ Bob\ left} : s}$$

Hence, as desired, Alice said Bob left is predicted to be an acceptable English utterance, in particular a sentence. Also, "said Bob left : s" has no proof, so *said Bob left is predicted to be unacceptable.

In this set of inference rules, every expression metavariable (A or B) that appears in the conclusion of a rule also appears in a premise of the rule. Thus the tree of rules used in a proof determines the final conclusion. For example, if we give a name to each rule in Figure 2.1, and add these names to (2.7), then we can omit all the judgments between and under the horizontal rules in (2.7) with no loss of information. This property, that the proof determines the expression, holds in many grammars below as well.

Because the judgment form and inference rules in Figure 2.1 are so simple and use only finitely many types, this grammar is just a context-free grammar in thin disguise: In more familiar notation, the grammar can be written as in Figure 2.2. Each type is a nonterminal symbol, each word is a terminal symbol, and each inference rule is a production.

Alas, the types in Figures 2.1 and 2.2 do not distinguish finely enough among expressions. For example, the grammar classifies both said and left as just verbs (v), so it predicts incorrectly that *Alice said Bob said is as acceptable as Alice said Bob left. We can begin to fix this problem by making finer distinctions

Expressions $A : T$

$$\frac{}{\mathsf{Alice} : np} \quad \frac{}{\mathsf{Bob} : np} \quad \frac{}{\mathsf{left} : iv} \quad \frac{}{\mathsf{slept} : iv} \quad \frac{}{\mathsf{saw} : tv} \quad \frac{}{\mathsf{loved} : tv} \quad \frac{}{\mathsf{said} : sv}$$

$$\frac{A : np \quad B : vp}{A\ B : s} \qquad \frac{A : iv}{A : vp} \qquad \frac{A : tv \quad B : np}{A\ B : vp} \qquad \frac{A : sv \quad B : s}{A\ B : vp}$$

Figure 2.3. A natural-language fragment that distinguishes among three kinds of verbs

$$np ::= \mathsf{Alice} \mid \mathsf{Bob} \quad iv ::= \mathsf{left} \mid \mathsf{slept} \quad tv ::= \mathsf{saw} \mid \mathsf{loved} \quad sv ::= \mathsf{said}$$
$$s ::= np\ vp \qquad vp ::= iv \mid tv\ np \mid sv\ s$$

Figure 2.4. The context-free grammar equivalent to Figure 2.3

in the types. Figures 2.3 and 2.4 show a refined system (as inference rules and as productions) that splits the type v into three more refined types of verbs: iv (intransitive verb), tv (transitive verb), and sv (sentential verb). This new system properly rules out *Alice said Bob said.

The linguistic rules so far rely on a binary concatenation operation on expressions (written with a space), so as to combine Bob and left to give Bob left. As we have been taking an expression to be a sequence of words (rather than a binary tree, a set, or a multiset of words), this concatenation operation is associative, non-commutative, and non-idempotent. The language models in the following sections treat the data structure of expressions more elaborately and explicitly: they allow more operations for building up expressions than just concatenation, and they take fewer properties of these operations for granted. One reason to explore such models is to go beyond context-free grammars. Context-free grammars are not ideal for modeling natural languages, for at least three reasons.

- Some natural languages are not context-free. For example, verb-object agreement in Swiss-German embedded clauses boils down (Shieber 1985) to the formal language of doubled strings $\{ ww \mid w \in \Sigma^* \}$, which is context-sensitive.
- Some natural-language phenomena are awkward to model with context-free grammars. In particular, many languages let a wh-phrase appear in front of a question, arbitrarily far from its "associated" verb. For example, who in the English question who do you think Alice said left is far from left.
- A context-free grammar only defines the syntax of a language and does

Expressions $A : s$

$$\frac{}{0 : s} \qquad \frac{A : s}{\mathsf{succ}\ A : s} \qquad \frac{A : s \quad B : s}{A + B : s} \qquad \frac{}{\mathsf{true} : s} \qquad \frac{}{\mathsf{false} : s} \qquad \frac{A : s \quad B : s}{A = B : s}$$

$$\frac{A : s \quad B_1 : s \quad B_2 : s}{\mathsf{if}\ A\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 : s}$$

Figure 2.5. A toy programming language for arithmetic

$$s ::= 0 \mid \mathsf{succ}\ s \mid s + s \mid \mathsf{true} \mid \mathsf{false} \mid s = s \mid \mathsf{if}\ s\ \mathsf{then}\ s\ \mathsf{else}\ s$$

Figure 2.6. The context-free grammar equivalent to Figure 2.5

not constrain its semantics. For example, although the expression Alice said Bob left is classified as a sentence, it is not required to have a meaning. Moreover, although the proof (2.7) splits into one branch for Alice and another for said Bob left, a meaning for Alice said Bob left need not be composed from one meaning for Alice and another for said Bob left. One may prefer to model natural language in a way that enforces a tighter correspondence between syntax and semantics.

We start to assuage these shortcomings in Section 2.4 below, with a framework for natural-language grammars that makes utterance structure more explicit and elaborate than just concatenation. But first, let us consider some programming languages.

2.2. A toy programming language

Figure 2.5 defines the syntax of a toy programming language. This grammar uses only one type s (for "start"), the type of programs. Accordingly, a judgment has the form $A : s$, pronounced "A is an s" or "A is a program", where A is an expression. (We omit "$: s$" from judgments in the next section for brevity.) Some expressions classified as programs are:

(2.8) succ 0 + succ succ 0 + succ succ succ 0,

(2.9) true + false,

(2.10) if true = succ succ 0 then false else 0 + 0,

(2.11) succ if true then false else succ 0.

As in Section 2.1, this grammar is equivalent to a context-free grammar, whose productions are shown in Figure 2.6.

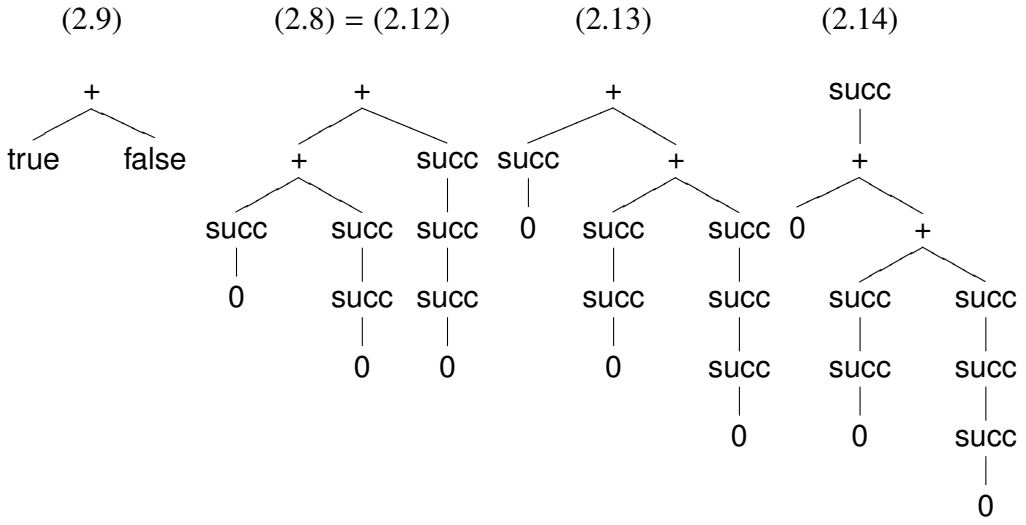


Figure 2.7. Expressions are trees but usually written with a textual representation

We take an expression in this language to be not a string of characters (*concrete syntax*) but a tree-like data structure (*abstract syntax*), such that

- each leaf is labeled by 0, true, or false;
- each unary branch is labeled by succ;
- each binary branch is labeled by + or =;
- each ternary branch is labeled by if-then-else; and
- there are no other kind of branches.

For example, the first tree in Figure 2.7 depicts graphically the same expression as (2.9) represents textually. This abstract syntax tree is equivalent to a context-free parse tree. It is also equivalent to a proof tree, so the tree of proof rules used determines the final conclusion judgment and the expression therein, as in previous grammars in Section 2.1.

Some concrete strings are ambiguous among many abstract expressions, which can be distinguished in writing using parentheses. For example, the string in (2.8) may be understood as any of the following expressions, among others.

(2.12) ((succ 0) + (succ succ 0)) + (succ succ succ 0)

(2.13) (succ 0) + ((succ succ 0) + (succ succ succ 0))

(2.14) succ (0 + ((succ succ 0) + (succ succ succ 0)))

(The remainder of Figure 2.7 depicts these three expressions graphically.) By convention, we let + and = associate to the left, so 0+0+0 represents (0+0)+0 and not 0+(0+0). Also by convention, we specify that succ has the highest *precedence*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{0} : s}}{\mathsf{succ\ 0} : s}}{\mathsf{succ\ succ\ 0} : s}}{\dfrac{\dfrac{}{\mathsf{true} : s} \quad \dfrac{\dfrac{}{\mathsf{succ\ succ\ 0} : s}}{\mathsf{true} = \mathsf{succ\ succ\ 0} : s}}{\mathsf{if\ true} = \mathsf{succ\ succ\ 0\ then\ false\ else\ 0 + 0} : s} \quad \dfrac{\dfrac{}{\mathsf{false} : s}}{\mathsf{false} : s} \quad \dfrac{\dfrac{}{\mathsf{0} : s} \quad \dfrac{}{\mathsf{0} : s}}{\mathsf{0 + 0} : s}}$$

Figure 2.8. Proving that if true = succ succ 0 then false else 0 + 0 is well formed

in the language, followed by +, =, and if-then-else (in that order). This way, the concrete string in (2.8) means only the abstract expression in (2.12). To take another example, the concrete string (2.10) means only the abstract expression

(2.15) if true = succ succ 0 then false else (0 + 0).

Figure 2.8 proves the well-formedness of this expression.

A popular use of programs is to run them. We now describe how to run a program in our toy language, using a *small-step operational semantics*. First we define the set of *evaluation contexts* (Felleisen 1987).[2] Intuitively, a context is an expression with a hole in it, where a subexpression can be plugged in. In other words, a context is an expression tree with a subtree removed, where a new subtree can be attached. For example,

(2.16) succ 0 + succ [ ] + succ succ succ 0

is a context, as is the trivial context [ ]. We write the metavariable $C[\ ]$ for a context. If $C[\ ]$ is a context and A is an expression, then we write $C[A]$ for the result of plugging A into $C[\ ]$. For example, if $C[\ ]$ is the context (2.16), then $C[\mathsf{succ\ 0}]$ is the expression (2.8) on page 24. If $C[\ ]$ and $C'[\ ]$ are two contexts, then we write $C[C'[\ ]]$ for the result of plugging $C'[\ ]$ into $C[\ ]$. Continuing the example, $C[\mathsf{0} = [\ ]]$ is the context

(2.17) succ 0 + succ (0 = [ ]) + succ succ succ 0.

Figure 2.9 uses this notation to define evaluation contexts formally. Our judgment form for evaluation contexts is simply $C[\ ]$, where $C[\ ]$ contains [ ]. This grammar

[2] For this toy language, it is overkill to specify the operational semantics in terms of evaluation contexts, but the notion of contexts is crucial for studying the computational side effect of delimited control and the linguistic side effect of quantification, which we start to do in Chapter 3.

Evaluation contexts $C[\ ]$

$$\frac{}{[\ ]} \qquad \frac{C[\ ]}{C[\mathsf{succ}\ [\ ]]} \qquad \frac{C[\ ] \quad B : s}{C[[\ ] + B]} \qquad \frac{A : s \quad C[\ ]}{C[A + [\ ]]} \qquad \frac{C[\ ] \quad B : s}{C[[\ ] = B]} \qquad \frac{A : s \quad C[\ ]}{C[A = [\ ]]}$$

$$\frac{C[\ ] \quad B_1 : s \quad B_2 : s}{C[\mathsf{if}\ [\ ]\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2]} \qquad \frac{A : s \quad C[\ ] \quad B_2 : s}{C[\mathsf{if}\ A\ \mathsf{then}\ [\ ]\ \mathsf{else}\ B_2]} \qquad \frac{A : s \quad B_1 : s \quad C[\ ]}{C[\mathsf{if}\ A\ \mathsf{then}\ B_1\ \mathsf{else}\ [\ ]]}$$

Figure 2.9. Evaluation contexts

$$\begin{aligned} c ::= {} & ; \mid c\ \mathsf{succ} \mid + s\ c \mid c\ s + \mid = s\ c \mid c\ s = \\ & \mid \mathsf{then}\ s\ \mathsf{else}\ s\ c\ \mathsf{if} \mid \mathsf{else}\ s\ c\ \mathsf{if}\ s\ \mathsf{then} \mid c\ \mathsf{if}\ s\ \mathsf{then}\ s\ \mathsf{else} \end{aligned}$$

Figure 2.10. The context-free grammar of evaluation contexts, equivalent to Figure 2.9

Computation $A \rhd B$

$$\begin{aligned} C[\mathsf{succ}^m 0 + \mathsf{succ}^n 0] &\rhd C[\mathsf{succ}^{m+n} 0] \\ C[\mathsf{succ}^m 0 = \mathsf{succ}^m 0] &\rhd C[\mathsf{true}] \\ C[\mathsf{succ}^m 0 = \mathsf{succ}^n 0] &\rhd C[\mathsf{false}] \qquad \text{if } m \neq n \\ C[\mathsf{if\ true\ then}\ B_1\ \mathsf{else}\ B_2] &\rhd C[B_1] \\ C[\mathsf{if\ false\ then}\ B_1\ \mathsf{else}\ B_2] &\rhd C[B_2] \end{aligned}$$

Figure 2.11. The computation relation between expressions. By $\mathsf{succ}^m 0$ we mean m copies of succ in front of 0, where m is a nonnegative integer.

can be viewed "inside-out" as a context-free grammar: if we change the notation of contexts to swap what is written to the left of the hole with what is written to the right, so that the context (2.16) is notated

(2.18) $$+\ \mathsf{succ\ succ\ succ\ 0}\ ;\ \mathsf{succ\ 0}\ +\ \mathsf{succ}$$

instead, then Figure 2.9 becomes a context-free grammar, shown in Figure 2.10.

Figure 2.11 introduces the *computation relation* $\rhd$, a binary relation between expressions. Intuitively, $A \rhd B$ holds just in case the expression A turns into the expression B in one small step of computation. Computation proceeds by chaining these small steps together. For example, starting with the expression (2.8) on

$$\begin{array}{ccc}
\mathsf{if\ 0 = 0\ then\ succ\ 0 + succ\ 0\ else\ false} & \triangleright & \mathsf{if\ 0 = 0\ then\ succ\ succ\ 0\ else\ false} \\
\downarrow & & \downarrow \\
\mathsf{if\ true\ then\ succ\ 0 + succ\ 0\ else\ false} & \triangleright & \mathsf{if\ true\ then\ succ\ succ\ 0\ else\ false} \\
\downarrow & & \downarrow \\
\mathsf{succ\ 0 + succ\ 0} & \triangleright & \mathsf{succ\ succ\ 0}
\end{array}$$

Figure 2.12. Three confluent ways to run the same program

page 24, we have

$$\begin{aligned}
&\mathsf{succ\ 0 + succ\ succ\ 0 + succ\ succ\ succ\ 0} \\
&\quad \triangleright \mathsf{succ\ succ\ succ\ 0 + succ\ succ\ succ\ 0} \\
&\quad \triangleright \mathsf{succ\ succ\ succ\ succ\ succ\ succ\ 0.}
\end{aligned} \tag{2.19}$$

We write $\triangleright^*$ for the reflexive and transitive closure of $\triangleright$, and $\triangleright^+$ for the transitive closure of $\triangleright$. Under the (intended) interpretation of succ as the successor function on numbers, this fact that

$$\begin{aligned}
&\mathsf{succ\ 0 + succ\ succ\ 0 + succ\ succ\ succ\ 0} \\
&\quad \triangleright^* \mathsf{succ\ succ\ succ\ succ\ succ\ succ\ 0}
\end{aligned} \tag{2.20}$$

means that (1 + 2) + 3 evaluates to 6.

The computation relation $\triangleright$ is nondeterministic: there exist expressions A such that $A \triangleright B_1$ and $A \triangleright B_2$ hold for two different expressions B_1 and B_2. For example, Figure 2.12 shows three ways to run the program at the upper-left corner to get the result at the lower-right corner. Nevertheless, this programming language is *confluent* (or equivalently, it has the *Church-Rosser* property): whenever $A \triangleright^* B_1$ and $A \triangleright^* B_2$ for some expressions A, B_1, and B_2, there exists an expression B' such that $B_1 \triangleright^* B'$ and $B_2 \triangleright^* B'$. This definition is illustrated below.

(2.21)

$$\begin{array}{ccc}
A & \longrightarrow^* & B_2 \\
\downarrow & & \vdots \\
B_1 & \cdots\cdots^* & B'
\end{array}$$

An expression A from which computation cannot proceed (that is, such that no B satisfies $A \triangleright B$) is said to be in *normal form*. Our language enjoys the *weak normalization* property, which means that every expression A has a normal form B such that $A \triangleright^* B$. In words, every program can be run in some way to stop. In fact, every way to run every program in this language eventually stops. That is, there

Expressions $A : T$

$$\frac{}{\mathsf{0} : n} \qquad \frac{A : n}{\mathsf{succ}\ A : n} \qquad \frac{A : n \quad B : n}{A + B : n} \qquad \frac{}{\mathsf{true} : b} \qquad \frac{}{\mathsf{false} : b} \qquad \frac{A : n \quad B : n}{A = B : b}$$

$$\frac{A : b \quad B_1 : T \quad B_2 : T}{\mathsf{if}\ A\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 : T}$$

Figure 2.13. A toy programming language with a more refined type system

does not exist an infinite sequence of computation steps $A_0 \triangleright A_1 \triangleright A_2 \triangleright \cdots$. This property is called *strong normalization* or *termination*.

Informally speaking, there are two kinds of expressions in normal form. On one hand, an expression in normal form may be the result of a successful computation, such as the last line in (2.19): succ succ succ succ succ succ 0. On the other hand, an expression in normal form may be stuck in an unsuccessful computation, such as (2.9) on page 24, which tries to add true to false. To distinguish between these two cases formally, we define a *value* to be either true, false, or zero or more copies of succ in front of 0. A normal form that is not a value, like (2.9), is *stuck*. A value is the result of a successful computation, whereas a stuck expression is the carnage of an unsuccessful computation.

The expressions (2.10) and (2.11) on page 24 are not initially stuck, but get stuck after some computation steps. In other words, these expressions never produce a value when run; they are akin to a program that begins running but then crashes. To rule out such errors without actually running the program, we can refine the type system of our language, as is done in Figure 2.13. The catch-all type s for programs becomes two types: n for (an expression that computes to) a number and b for (an expression that computes to) a Boolean. A judgment in this type system has the form $A : T$, where A is a metavariable that ranges over expressions, and T is a metavariable that ranges over types. Types are generated by the simple context-free grammar

$$T ::= n \mid b. \tag{2.22}$$

This type system shares with the previous one in Figure 2.5 the *subject reduction* property, also known as *type preservation*: if $A \triangleright B$ and $A : T$, then $B : T$ as well. Furthermore, no expression to which this refined type system assigns a type (either n or b) is stuck. That is, any expression assigned a type either is a value or computes to some expression. These two properties together entail that any expression assigned a type never computes to a stuck expression—in other words, never "goes wrong". Formally, if $A \triangleright^* B$ and $A : T$, then B is not stuck. In particular, the refined type system assigns no type to ("rules out") the expressions

(2.10) and (2.11).

For the purpose of ruling out programs that go wrong, this type system only classifies expressions approximately. Besides ruling out every program that goes wrong, the system also rules out some programs (hopefully seldom encountered or needed in practice) that do not go wrong when run. For example, the program (similar to (2.11) on page 24)

(2.23) $\quad$ succ if false then false else succ 0

is no longer assigned a type according to Figure 2.13, yet computes to a value just fine:

(2.24) $\quad$ succ if false then false else succ 0 $\triangleright$ succ succ 0.

This kind of approximation is often necessary in a practical type system, especially for a programming language without the termination property, because it may take too long or even be impossible for a computer to decide precisely whether a given program goes wrong. Even when every computation sequence is finite, as is the case here, an approximate type serves to summarize an expression to a programmer, much as a natural-language type classifies an expression to a linguist.

2.3. The λ-calculus

A crucial ability missing from the toy programming language in Section 2.2 is to pass a program or its result for use elsewhere. The *λ-calculus* (Church 1932, 1940; Barendregt 1981; Hindley and Seldin 1986) is a canonical programming language that incorporates this ability by allowing values, including functions, to be passed around. Because the λ-calculus lets us pass functions as values, it is known as a *higher-order* programming language.

We first introduce the λ-calculus informally, then consider its type systems and operational semantics. The three kinds of expressions in the λ-calculus are *variables*, *abstractions*, and *applications*. The expression $\lambda x.\,\mathsf{succ}\ 0 + x$ is an abstraction; it denotes a function that maps each number to its successor. The variable x here, representing the input to the function, is *bound* within the *body* $\mathsf{succ}\ 0 + x$ of the abstraction. We notate the application of a function F to an argument E as FE. A function can apply multiple times to different arguments, each time binding the variable to a different input. For example, the application expression

(2.25) $\quad (\lambda f.\, f(0) + f(\mathsf{succ}\ \mathsf{succ}\ 0))(\lambda x.\,\mathsf{succ}\ 0 + x)$

first binds f to the increment function, then binds x to 0 in the first application of f, but to succ succ 0 in the second application of f. This program thus computes $(1 + 0) + (1 + 2)$.

Expressions $\Gamma \vdash E$

$$\frac{}{x \vdash x}\ \text{Id} \qquad \frac{\Gamma, x \vdash E}{\Gamma \vdash \lambda x.\, E}\ \text{Abstract} \qquad \frac{\Gamma \vdash F \quad \Delta \vdash E}{\Gamma, \Delta \vdash FE}\ \text{Apply}$$

$$\frac{\Gamma[\Delta] \vdash E}{\Gamma[\Delta, \Theta] \vdash E}\ \text{Weaken} \qquad \frac{\Gamma[\Delta, \Delta] \vdash E}{\Gamma[\Delta] \vdash E}\ \text{Contract}$$

$$\frac{\Gamma[\Delta, \Theta] \vdash E}{\Gamma[\Theta, \Delta] \vdash E}\ \text{Exchange} \qquad \frac{\Gamma[(\Delta, \Theta), \Pi] \vdash E}{\Gamma[\Delta, (\Theta, \Pi)] \vdash E}\ \text{Associate}$$

Figure 2.14. The untyped λ-calculus

For brevity, we omit some parentheses in expressions: Abstractions extend as far to the right as possible, so $\lambda x.\, xy$ means $\lambda x.\, (xy)$ rather than $(\lambda x.\, x)y$. Applications associate to the left, so xyz means $(xy)z$ rather than $x(yz)$.

By convention, when multiple nested abstractions bind the same variable name, the innermost binding "wins". For example, the expression $\lambda x.\, \lambda x.\, x$ means the constant function returning the identity function—that is, $\lambda y.\, \lambda x.\, x$ rather than $\lambda x.\, \lambda y.\, x$. Expressions like $\lambda x.\, \lambda x.\, x$ and $\lambda y.\, \lambda x.\, x$ are *α-equivalent* in that they (informally speaking) differ only in the names of bound variables. We henceforth regard α-equivalent expressions to be equal. We then assume that all bound variables in an expression are renamed whenever necessary to be distinct from each other; this assumption is Barendregt's *variable convention* (1981). In other words, bound-variable names are not part of the abstract syntax of this language; rather, they are just nomenclature for associating an abstraction expression with variable expressions in its body.

To *substitute* an expression E for a variable x in another expression E' is to replace every occurrence of x in E' by E. We notate the result as $E'\{x \mapsto E\}$. (Here x is a metavariable that ranges over variables in the λ-calculus.) For example,

$$(x(\lambda y.\, x))\{x \mapsto \lambda z.\, zw\} = (\lambda z.\, zw)(\lambda y.\, \lambda z.\, zw). \tag{2.26}$$

Because the same variable may be bound multiple times in E and E', substitution is not just a matter of textual replacement, but may involve renaming variables to avoid conflicts. For example,

$$(x(\lambda y.\, x))\{x \mapsto \lambda x.\, yx\} = (\lambda x.\, yx)(\lambda y'.\, \lambda x.\, yx) \neq (\lambda x.\, yx)(\lambda y.\, \lambda x.\, yx). \tag{2.27}$$

Figure 2.14 defines the *untyped λ-calculus*, which is so called because it only classifies expressions as well-formed. We use E and F as metavariables for expressions. To keep track of bound variables, judgments in this system are of the form $\Gamma \vdash E$. Here the *type environment* (or *antecedent*) Γ is a list of bound

variables—more precisely, a binary tree whose leaves are either just the null environment, written with a dot $\cdot$, or an *assumption* x, meaning that the variable x is bound. The *turnstile* $\vdash$ separates the type environment from the expression E classified well-formed. Besides the metavariable Γ for type environments, we also use the Greek letters Δ, Θ, Π.

The first three rules in this system construct variables, abstractions, and applications. The Id rule says that a variable is a well-formed expression ($\cdots \vdash x$), as long as it is bound ($x \vdash \cdots$). The Abstract rule says that an abstraction $\lambda x.\, E$ is well formed in the type environment Γ, as long as the body E is well formed in Γ extended with an assumption for the bound variable x. The Apply rule says that, if F and E are well formed in the type environments Γ and Δ, then the application FE is well formed in the combined type environment Γ, Δ.

As with expressions, we equate judgments that differ only in the name of bound variables. A variable name in an environment, like the variable name bound by an abstraction expression, is just nomenclature for associating an assumption to the left of $\vdash$ with variable expressions to the right of $\vdash$. We then assume that all bound variables in a judgment are renamed whenever necessary to be distinct from each other. Thus the Abstract rule in Figure 2.14 is really shorthand for a more elaborate rule that renames the bound variable x to a freshly chosen name y:

$$\frac{\Gamma,\, y \vdash E\{x \mapsto y\} \quad E \text{ does not mention } y}{\Gamma \vdash \lambda x.\, E}\ \text{Abstract.} \tag{2.28}$$

The remaining four rules in the system are *structural*. They ensure that a type environment behaves as a set of bound variables, even though strictly speaking it has the structure of a binary tree. The notation $\Gamma[\ldots]$ in these rules means a type environment that includes another. For example, $\Gamma[\Delta, \Delta]$ in the Contract rule matches any type environment that somewhere contains two identical type environments Δ together. Accordingly, $\Gamma[\Delta]$ in the same rule matches the same type environment with only one copy of Δ at the same place. Thus $\Gamma[\,]$ is a metavariable for not a complete type environment but a type environment with a hole $[\,]$.

The Weaken rule says that it never hurts the well-formedness of an expression to assume an extra bound variable; that is, a bound variable's value can be discarded freely. The Contract rule says that duplicate assumptions in a type environment can be merged; that is, a bound variable's value can be duplicated freely. The Exchange rule says that order does not matter in a type environment; that is, the comma for combining type environments is commutative. The Associate rule is written with a special notation: the double horizontal rule between the two judgments means that either judgment can be used as the premise in an inference, and the other judgment as the conclusion. This rule says that the comma for combining type environments is associative. If we include the antecedents of each

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f \vdash f}\,\text{Id}}{f, \cdot \vdash f}\,\text{Weaken}}{\cdot, f \vdash f}\,\text{Exchange} \quad \dfrac{\dfrac{}{f \vdash f}\,\text{Id} \quad \dfrac{}{x \vdash x}\,\text{Id}}{f, x \vdash fx}\,\text{Apply}}{(\cdot, f), (f, x) \vdash f(fx)}\,\text{Apply}}{((\cdot, f), f), x \vdash f(fx)}\,\text{Associate}}{(\cdot, f), f \vdash \lambda x.\, f(fx)}\,\text{Abstract}}{\cdot, (f, f) \vdash \lambda x.\, f(fx)}\,\text{Associate}}{\cdot, f \vdash \lambda x.\, f(fx)}\,\text{Contract}}{\cdot \vdash \lambda f.\, \lambda x.\, f(fx)}\,\text{Abstract}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f \vdash f}\,\text{Id} \quad \dfrac{\dfrac{}{f \vdash f}\,\text{Id} \quad \dfrac{}{x \vdash x}\,\text{Id}}{f, x \vdash fx}\,\text{Apply}}{f, x \vdash f(fx)}\,\text{Apply}}{f \vdash \lambda x.\, f(fx)}\,\text{Abstract}}{\cdot \vdash \lambda f.\, \lambda x.\, f(fx)}\,\text{Abstract}$$

Figure 2.15. Proving that $\lambda f.\, \lambda x.\, f(fx)$ is well formed

Evaluation contexts $C[\ ]$

$$\frac{}{[\ ]} \qquad \frac{C[\ ]}{C[\lambda x.\, [\ ]]} \qquad \frac{C[\ ] \quad \Delta \vdash E}{C[[\ ]E]} \qquad \frac{\Gamma \vdash F \quad C[\ ]}{C[F[\ ]]}$$

Figure 2.16. Evaluation contexts in the λ-calculus

structural rule in its name—for example, if we distinguish among instances of Weaken with different Γ or Θ—then this grammar, like previous ones, ensures that the tree of rules used in a proof determines the final judgment.

To illustrate all these rules, Figure 2.15 derives that the "twice" function $\lambda f.\, \lambda x.\, f(fx)$ is a well-formed expression. The same proof is shown there twice: first in its full glory, regarding the type environment as a binary tree, and then with the structural inferences omitted for brevity, regarding the type environment as a set of bound variables rather than a tree.

A program in the λ-calculus runs as follows. Figure 2.16 defines a judgment form $C[\ ]$, which means that $C[\ ]$ is a context that can be plugged with some expression E to make a complete program $C[E]$. The context $C[\ ]$ may bind variables in E; for example, plugging the expression $E = fx$ into the context $C[\ ] = \lambda f.\, \lambda x.\, f[\ ]$ makes the complete program $C[E] = \lambda f.\, \lambda x.\, f(fx)$. Armed with this definition of an evaluation context, we can then define a computation relation $\triangleright$, called *β-reduction* or *λ-conversion*:

$$C[(\lambda x.\, E')E] \triangleright C[E'\{x \mapsto E\}]. \tag{2.29}$$

Here $C[\ ]$ is any evaluation context, and E and E' are any expressions, such that the program on the left is well formed, in which case the program on the right

is also well formed. As before, we write $\rhd^*$ to mean the reflexive and transitive closure of this computation relation, and $\rhd^+$ to mean the transitive closure. The subject reduction property holds: if $E \rhd^* E'$ and $\Gamma \vdash E$, then $\Gamma \vdash E'$ as well.

To illustrate β-reduction, let us apply the "twice" function (from Figure 2.15) to itself, to get the "four times" function.

$$\begin{aligned}(2.30)\qquad & (\lambda f.\lambda x.\, f(fx))(\lambda f.\lambda x.\, f(fx)) \\ & \rhd \lambda x.\, (\lambda f.\lambda x.\, f(fx))((\lambda f.\lambda x.\, f(fx))x) \\ & = \lambda f.\, (\lambda f.\lambda x.\, f(fx))((\lambda f.\lambda x.\, f(fx))f) \quad \text{(by } \alpha\text{-equivalence)} \\ & \rhd \lambda f.\, (\lambda f.\lambda x.\, f(fx))(\lambda x.\, f(fx)) \\ & \rhd \lambda f.\lambda x.\, (\lambda x.\, f(fx))((\lambda x.\, f(fx))x) \\ & \rhd \lambda f.\lambda x.\, (\lambda x.\, f(fx))(f(fx)) \\ & \rhd \lambda f.\lambda x.\, f(f(f(fx)))\end{aligned}$$

To take another example, if we enrich the syntax of this λ-calculus to encompass arithmetic expressions like (2.25) on page 30, then we can perform the following computation.

$$\begin{aligned}(2.31)\qquad & (\lambda f.\, f(0) + f(\mathsf{succ}\ \mathsf{succ}\ 0))(\lambda x.\, \mathsf{succ}\ 0 + x) \\ & \rhd (\lambda x.\, \mathsf{succ}\ 0 + x)(0) + (\lambda x.\, \mathsf{succ}\ 0 + x)(\mathsf{succ}\ \mathsf{succ}\ 0) \\ & \rhd (\lambda x.\, \mathsf{succ}\ 0 + x)(0) + (\mathsf{succ}\ 0 + \mathsf{succ}\ \mathsf{succ}\ 0) \\ & \rhd (\mathsf{succ}\ 0 + 0) + (\mathsf{succ}\ 0 + \mathsf{succ}\ \mathsf{succ}\ 0)\end{aligned}$$

Via a suitably enlarged computation relation, the last expression above may compute eventually to $\mathsf{succ}\ \mathsf{succ}\ \mathsf{succ}\ \mathsf{succ}\ 0$, as desired.[3]

For clarity in the many example programs presented below in the λ-calculus, we sometimes write

$$(2.32)\qquad \mathsf{let}\ E\ \mathsf{be}\ x.\, E'$$

to mean $(\lambda x.\, E')E$.[4] This syntactic sugar makes sense because of β-reduction. We call E the argument of the let, and E' the body of the let. By convention, the body E' of the let expression extends as far to the right as possible, like the body of the corresponding λ-abstraction.

[3] Many programming examples below assume, like this one, that the syntax and computation relation of the programming language are suitably extended with integer arithmetic as in Section 2.2. We do not formalize such *constants* as succ, 0, and + and computation steps for them, because they are not needed for the technical results in this dissertation. If desired, one can implement them using products, unit, sums, and recursion, described in Sections 2.3.3, 2.3.4, and 2.3.5.

[4] This notation orders E and x differently from the usual syntax $\mathsf{let}\ x = E\ \mathsf{in}\ E'$ in functional programming languages. We choose this notation so that we can extend it to perform case discrimination ($\mathsf{case}\ E\ \mathsf{of}\ \ldots$ in typical functional programming languages) in Section 2.3.4.

Expressions $\Gamma \vdash E : T$

$$\frac{}{x : T \vdash x : T}\ \text{Id} \qquad \frac{\Gamma,\, x : T_1 \vdash E : T_2}{\Gamma \vdash \lambda x.\, E : T_1 \to T_2}\ \text{Abstract}$$

$$\frac{\Gamma \vdash F : T_1 \to T_2 \quad \Delta \vdash E : T_1}{\Gamma, \Delta \vdash FE : T_2}\ \text{Apply}$$

$$\frac{\Gamma[\Delta] \vdash E : T}{\Gamma[\Delta, \Theta] \vdash E : T}\ \text{Weaken} \qquad \frac{\Gamma[\Delta, \Delta] \vdash E : T}{\Gamma[\Delta] \vdash E : T}\ \text{Contract}$$

$$\frac{\Gamma[\Delta, \Theta] \vdash E : T}{\Gamma[\Theta, \Delta] \vdash E : T}\ \text{Exchange} \qquad \frac{\Gamma[(\Delta, \Theta), \Pi] \vdash E : T}{\Gamma[\Delta, (\Theta, \Pi)] \vdash E : T}\ \text{Associate}$$

Evaluation contexts $C[\ ]$

$$\frac{}{[\ ]} \qquad \frac{C[\ ]}{C[\lambda x.\, [\ ]]} \qquad \frac{C[\ ] \quad \Delta \vdash E : T_1}{C[[\ ]E]} \qquad \frac{\Gamma \vdash F : T_0 \quad C[\ ]}{C[F[\ ]]}$$

Figure 2.17. The simply-typed λ-calculus

2.3.1. Confluence, normalization, and typing. The computation relation just defined is confluent (Barendregt 1981). (As we extend this programming language in Chapter 3, confluence becomes threatened.) But it is not weakly normalizing, let alone strongly normalizing. For example, the expression

$$(\lambda x.\, xx)(\lambda x.\, xx) \tag{2.33}$$

β-reduces to itself and no other expression, so there is no normal-form expression E such that $(\lambda x.\, xx)(\lambda x.\, xx) \rhd^* E$. Informally speaking, the program (2.33) enters an infinite loop when run.

To enforce normalization, we can refine the type system. We classify expressions into a countably infinite family of types, not just well-formed or *s*. A type is now either an element of a fixed set of *base types*, say $a, b, c, \ldots$, or a *function type*, written $T_1 \to T_2$ where T_1 and T_2 are again types. More formally, types T are generated by the context-free grammar

$$T ::= A \mid T \to T, \tag{2.34}$$

where base types A are generated by

$$A ::= a \mid b \mid c \mid \cdots . \tag{2.35}$$

A type environment Γ no longer just enumerates bound variables but also associates each variable with a type. Figure 2.17 shows the refined type system, called the *simply-typed* λ-calculus. For example, Figure 2.18 proves that the "twice"

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f : a \to a \vdash f : a \to a}\,\text{Id}}{f : a \to a \vdash f : a \to a}\,\text{Id} \quad \dfrac{\dfrac{}{f : a \to a \vdash f : a \to a}\,\text{Id} \quad \dfrac{}{x : a \vdash x : a}\,\text{Id}}{f : a \to a,\, x : a \vdash fx : a}\,\text{Apply}}{f : a \to a,\, x : a \vdash f(fx) : a}\,\text{Apply}}{f : a \to a \vdash \lambda x.\, f(fx) : a \to a}\,\text{Abstract}}{\cdot \vdash \lambda f.\, \lambda x.\, f(fx) : (a \to a) \to (a \to a)}\,\text{Abstract}$$

Figure 2.18. Proving that $\lambda f.\, \lambda x.\, f(fx)$ has type $(a \to a) \to (a \to a)$

function $\lambda f.\, \lambda x.\, f(fx)$ has the type $(a \to a) \to (a \to a)$. (We omit the structural rules in this proof; they are entirely analogous to those in Figure 2.15.)

By convention, the function-type arrow $\to$ associates to the right. For example, $a \to b \to c$ means the type $a \to (b \to c)$ rather than $(a \to b) \to c$.

Some expressions in the untyped λ-calculus can no longer be derived in the simply-typed λ-calculus. For example, the infinite-loop program (2.33) cannot be assigned a type in the refined system. The computation relation $\triangleright$ for the simply-typed λ-calculus is just the restriction of the computation relation for the untyped λ-calculus to those expressions that can still be derived in the refined system. As in Section 2.2, the refined system preserves the subject reduction property: if $E \triangleright^* E'$ and $\Gamma \vdash E : T$, then $\Gamma \vdash E' : T$ as well. Furthermore, every simply-typed λ-expression is strongly normalizing. That is, whenever $\Gamma \vdash E : T$ for some environment Γ, expression E and type T, there is no infinite sequence of computation steps $E \triangleright E_1 \triangleright E_2 \triangleright \cdots$ starting at E.

We have been using the word "function" loosely in reference to expressions and types. Technically speaking, expressions in the untyped λ-calculus are not—cannot be interpreted as—functions to be applied in the traditional, set-theoretic sense. For example, although we can think of the unique expression $\lambda x.\, x$ informally as "the identity function", there is no such thing as *the* identity function in the traditional, set-theoretic sense—only a family of identity functions, one for each domain. By contrast, an expression in the simply-typed λ-calculus can be interpreted set-theoretically, provided that we fix the type of the expression and specify a set for each base type mentioned in that type.

2.3.2. The formulas-as-types correspondence. In the simply-typed λ-calculus, as with previous grammars, the tree of inference rules used in a proof determines the final conclusion of the proof (provided, as before, that instances of a structural rule with different antecedents count as different inference rules). We can thus omit expressions and variables from judgments with no loss of information.

Figure 2.19 shows the result of omitting expressions and variables from the

$$\frac{}{T \vdash T}\,\mathrm{Id} \qquad \frac{\Gamma, T_1 \vdash T_2}{\Gamma \vdash T_1 \to T_2} \to \mathrm{I} \qquad \frac{\Gamma \vdash T_1 \to T_2 \quad \Delta \vdash T_1}{\Gamma, \Delta \vdash T_2} \to \mathrm{E}$$

$$\frac{\Gamma[\Delta] \vdash T}{\Gamma[\Delta, \Theta] \vdash T}\,\mathrm{Weaken} \qquad \frac{\Gamma[\Delta, \Delta] \vdash T}{\Gamma[\Delta] \vdash T}\,\mathrm{Contract}$$

$$\frac{\Gamma[\Delta, \Theta] \vdash T}{\Gamma[\Theta, \Delta] \vdash T}\,\mathrm{Exchange} \qquad \frac{\Gamma[(\Delta, \Theta), \Pi] \vdash T}{\Gamma[\Delta, (\Theta, \Pi)] \vdash T}\,\mathrm{Associate}$$

Figure 2.19. Natural-deduction rules for propositional intuitionistic logic

$$\dfrac{\dfrac{\dfrac{}{a \to a \vdash a \to a}\,\mathrm{Id} \quad \dfrac{\dfrac{}{a \to a \vdash a \to a}\,\mathrm{Id} \quad \dfrac{}{a \vdash a}\,\mathrm{Id}}{a \to a, a \vdash a} \to \mathrm{E}}{a \to a, a \vdash a} \to \mathrm{E}}{\dfrac{a \to a \vdash a \to a}{\cdot \vdash (a \to a) \to (a \to a)} \to \mathrm{I}} \to \mathrm{I}$$

Figure 2.20. Proving that $(a \to a) \to (a \to a)$

simply-typed λ-calculus. This "grammar without expressions" turns out to be a system for propositional intuitionistic logic, as follows. We read $a, b, c, \ldots$ as not base types but atomic formulas, and $T_1 \to T_2$ as not the function type from T_1 to T_2 but the implication formula from T_1 to T_2. Under this interpretation, the inference rule named Abstract in Figure 2.17 and $\to$I in Figure 2.19 is hypothetical reasoning (implication introduction), and the rule named Apply in Figure 2.17 and $\to$E in Figure 2.19 is modus ponens (implication elimination). The antecedent, to the left of $\vdash$, lists the hypotheses that have yet to be discharged. The structural rules enforce the fact that hypotheses in intuitionistic logic form an unordered set, from which an unused hypothesis can be freely discarded.

For example, Figure 2.20 (the result of erasing expressions from Figure 2.18) proves the formula $(a \to a) \to (a \to a)$ in intuitionistic logic. Conversely, the λ-calculus expression $\lambda f.\, \lambda x.\, f(fx)$ is a succinct way to write this proof that saves space and hides where structural rules are used. This proof is one of an infinite number of proofs of the same formula:

$$\lambda f.\, \lambda x.\, x, \quad \lambda f.\, \lambda x.\, fx, \quad \lambda f.\, \lambda x.\, f(fx), \quad \lambda f.\, \lambda x.\, f(f(fx)), \quad \ldots. \tag{2.36}$$

Propositional intuitionistic logic thus corresponds to the simply-typed λ-calculus: Formulas are types (atomic formulas are base types; implication formulas are function types). Proofs are programs (the trivial proof is a variable; to compose

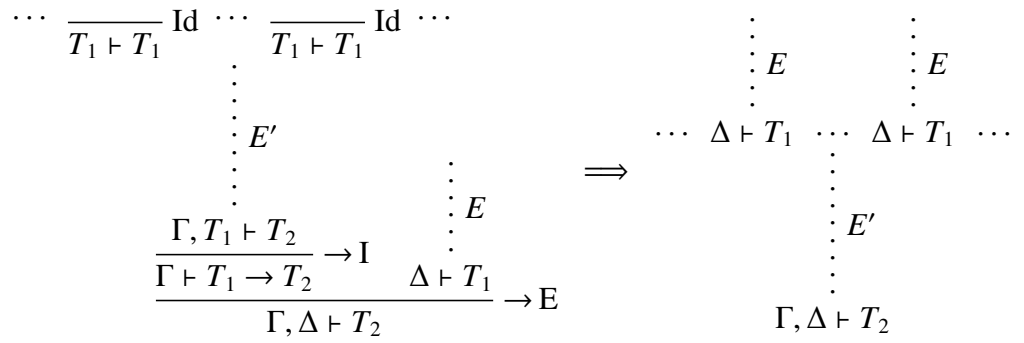


Figure 2.21. Proof reduction for $\to$I followed by $\to$E

two programs is to sequence two proofs). Moreover, to reduce a proof is to execute a program: to β-reduce a program, as specified in (2.29) on page 33, is to eliminate the detour in a proof that occurs when the conclusion of $\to$I is used as the first premise of $\to$E, as shown in Figure 2.21. The bound variable x in a λ-abstraction $\lambda x.\, E'$ represents a hypothesized proof; to β-reduce the program $(\lambda x.\, E')E$ is to substitute the actual proof E for x in E'. To emphasize this correspondence, we henceforth take Abstract to be synonymous with $\to$I, and Apply to be synonymous with $\to$E.

This *formulas-as-types correspondence* is also called the *Curry-Howard isomorphism* (Howard 1980; Girard et al. 1989; Wadler 2000). We use it in both directions. First, any semantics for the simply-typed λ-calculus or a variation on it, such as the usual semantics in which types denote sets and λ-abstractions denote functions, is a semantics for propositional intuitionistic logic or a (corresponding) variation on it. In Section 2.4 below, we present a formal system for natural-language grammars that is a variation on propositional intuitionistic logic, and endow that system with just such a semantics: we turn the proof that an utterance is grammatical into the meaning of that utterance (van Benthem 1983, 1988; Carpenter 1997; Ranta 1994). Our linguistic formalism thus extends the formulas-as-types correspondence to natural language: formulas and types are syntactic categories; proofs and programs are utterance meanings.

Second, extensions to propositional intuitionistic logic guide us to extend the simply-typed λ-calculus. We now follow this guidance to extend the simply-typed λ-calculus with additional data structures besides functions. These extensions yield a more practical programming language that is suitable for subsequent example programs.

2.3.3. Products. Besides functions, we can add ordered pairs to our language, to help build data structures with multiple components. A *product* type

Expressions $\Gamma \vdash E : T$ (additional)

$$\frac{\Gamma \vdash E_1 : T_1 \quad \Delta \vdash E_2 : T_2}{\Gamma, \Delta \vdash \langle E_1, E_2 \rangle : T_1 \times T_2} \times \mathrm{I} \qquad \frac{\Delta \vdash E : T_1 \times T_2 \quad \Gamma[x : T_1, y : T_2] \vdash E' : T'}{\Gamma[\Delta] \vdash \mathsf{let}\ E\ \mathsf{be}\ \langle x, y \rangle.\ E' : T'} \times \mathrm{E}$$

Evaluation contexts $C[\ ]$ (additional)

$$\frac{C[\ ] \quad \Delta \vdash E_2 : T_2}{C[\langle [\ ], E_2 \rangle]} \qquad \frac{\Gamma \vdash E_1 : T_1 \quad C[\ ]}{C[\langle E_1, [\ ] \rangle]}$$

$$\frac{C[\ ] \quad \Gamma \vdash E' : T'}{C[\mathsf{let}\ [\ ]\ \mathsf{be}\ \langle x, y \rangle.\ E']} \qquad \frac{\Delta \vdash E : T_0 \quad C[\ ]}{C[\mathsf{let}\ E\ \mathsf{be}\ \langle x, y \rangle.\ [\ ]]}$$

Computation $E \triangleright E'$ (additional)

$$C[\mathsf{let}\ \langle E_1, E_2 \rangle\ \mathsf{be}\ \langle x, y \rangle.\ E'] \triangleright C[E'\{x \mapsto E_1\}\{y \mapsto E_2\}]$$

Figure 2.22. Adding products to the λ-calculus

$T_1 \times T_2$ classifies an ordered pair of something of type T_1 and something of type T_2. For example, assuming that $\mathsf{2}$ has the type int, the expression $\langle \mathsf{2}, (\lambda f.\ f)(\lambda x.\ x) \rangle$ is an ordered pair with the type int $\times$ (int $\to$ int). To extract the two components of this pair, we can write code like

$$\mathsf{let}\ \langle \mathsf{2}, (\lambda f.\ f)(\lambda x.\ x) \rangle\ \mathsf{be}\ \langle x, y \rangle.\ yx, \tag{2.37}$$

which computes either to

$$((\lambda f.\ f)(\lambda x.\ x))(\mathsf{2}) \tag{2.38}$$

or to

$$\mathsf{let}\ \langle \mathsf{2}, \lambda x.\ x \rangle\ \mathsf{be}\ \langle x, y \rangle.\ yx, \tag{2.39}$$

both of which in turn compute to $(\lambda x.\ x)\mathsf{2}$, and from there to $\mathsf{2}$. In an expression like $\mathsf{let}\ E\ \mathsf{be}\ \langle x, y \rangle.\ E'$, we call E the argument and E' the body.

Figure 2.22 formalizes the two rules for expressions, four rules for contexts, and one rule for computation that we just illustrated. Types T are now generated by the context-free grammar

$$T ::= A \mid T \to T \mid T \times T, \tag{2.40}$$

with a new case $T \times T$ at the end.

Via the formulas-as-types correspondence, a product type (written here with the connective $\times$) corresponds to a conjunction formula (written typically with the connective $\wedge$), and a program of a product type corresponds to a proof of a conjunction formula. Similarly, corresponding to logical truth (verum, typically written $\top$) is the *unit* or *singleton* type (here written 1). The unit type 1 contains

Expressions $\Gamma \vdash E : T$ (additional)

$$\frac{}{\cdot \vdash \langle\rangle : 1}\,1\,\mathrm{I} \qquad \frac{\Delta \vdash E : 1 \quad \Gamma[\cdot] \vdash E' : T'}{\Gamma[\Delta] \vdash \mathsf{let}\ E\ \mathsf{be}\ \langle\rangle.\ E' : T'}\,1\,\mathrm{E}$$

Evaluation contexts $C[\ ]$ (additional)

$$\frac{C[\ ] \quad \Gamma \vdash E' : T'}{C[\mathsf{let}\ [\ ]\ \mathsf{be}\ \langle\rangle.\ E']} \qquad \frac{\Delta \vdash E : T_0 \quad C[\ ]}{C[\mathsf{let}\ E\ \mathsf{be}\ \langle\rangle.\ [\ ]]}$$

Computation $E \triangleright E'$ (additional)

$$C[\mathsf{let}\ \langle\rangle\ \mathsf{be}\ \langle\rangle.\ E'] \triangleright C[E']$$

Figure 2.23. Adding unit to the λ-calculus

just one element, denoted by the expression $\langle\rangle$. It is useful sometimes as a placeholder. Figure 2.23 formalizes the rules. Types T are now generated by the context-free grammar

$$T ::= A \mid T \to T \mid T \times T \mid 1, \tag{2.41}$$

with a new case 1 at the end.

2.3.4. Sums. We can also add disjoint unions of types to our programming language. A *sum* type $T_1 + T_2$ classifies something of type T_1 or type T_2, discriminated using labels left and right. For example, if 2 has the type int, then the expressions $\mathsf{left}\,2$ and $\mathsf{right}((\lambda f.\ f)(\lambda z.\ z))$ both have the type int + (int → int). Given something of sum type $T_1 + T_2$, we want to check whether it is a T_1 labeled with left or a T_2 labeled with right, and to extract the labeled T_1 or T_2. To do so, we can write code like

$$\mathsf{let}\ \mathsf{left}\,2\ \mathsf{be}\ \mathsf{left}\,x.\ x \mid \mathsf{right}\,y.\ y5 \tag{2.42}$$

(which chooses the left alternative) and

$$\mathsf{let}\ \mathsf{right}((\lambda f.\ f)(\lambda z.\ z))\ \mathsf{be}\ \mathsf{left}\,x.\ x \mid \mathsf{right}\,y.\ y5 \tag{2.43}$$

(which chooses the right alternative). These programs compute to 2 and 5, respectively.

Another example, perhaps a more obviously useful one, is to encode the Boolean type as 1 + 1: the sum of two unit types. We can represent, say, false as $\mathsf{left}\langle\rangle$ and true as $\mathsf{right}\langle\rangle$, both of type 1 + 1 (but distinct, since we are taking the *disjoint* union of 1 and 1). If x and y are both Booleans encoded in this way, of type 1 + 1, then their conjunction is

$$\mathsf{let}\ x\ \mathsf{be}\ \mathsf{left}\,u.\ \mathsf{left}\langle\rangle \mid \mathsf{right}\,u.\ y, \tag{2.44}$$

Expressions $\Gamma \vdash E : T$ (additional)

$$\frac{\Gamma \vdash E : T_1}{\Gamma \vdash \mathsf{left}\, E : T_1 + T_2}\,{+}\mathrm{I} \qquad \frac{\Gamma \vdash E : T_2}{\Gamma \vdash \mathsf{right}\, E : T_1 + T_2}\,{+}\mathrm{I}$$

$$\frac{\Delta \vdash E : T_1 + T_2 \quad \Gamma[x : T_1] \vdash E_1' : T' \quad \Gamma[y : T_2] \vdash E_2' : T'}{\Gamma[\Delta] \vdash \mathsf{let}\, E\, \mathsf{be\ left}\, x.\, E_1' \mid \mathsf{right}\, y.\, E_2' : T'}\,{+}\mathrm{E}$$

Evaluation contexts $C[\ ]$ (additional)

$$\frac{C[\ ]}{C[\mathsf{left}\,[\ ]]} \qquad \frac{C[\ ]}{C[\mathsf{right}\,[\ ]]} \qquad \frac{C[\ ] \quad \Gamma_1 \vdash E_1' : T_1' \quad \Gamma_2 \vdash E_2' : T_2'}{C[\mathsf{let}\,[\ ]\, \mathsf{be\ left}\, x.\, E_1' \mid \mathsf{right}\, y.\, E_2']}$$

$$\frac{\Delta \vdash E : T_0 \quad C[\ ] \quad \Gamma \vdash E_2' : T'}{C[\mathsf{let}\, E\, \mathsf{be\ left}\, x.\, [\ ] \mid \mathsf{right}\, y.\, E_2']} \qquad \frac{\Delta \vdash E : T_0 \quad \Gamma \vdash E_1' : T' \quad C[\ ]}{C[\mathsf{let}\, E\, \mathsf{be\ left}\, x.\, E_1' \mid \mathsf{right}\, y.\, [\ ]]}$$

Computation $E \rhd E'$ (additional)

$$C[\mathsf{let\ left}\, E\, \mathsf{be\ left}\, x.\, E_1' \mid \mathsf{right}\, y.\, E_2'] \rhd C[E_1'\{x \mapsto E\}]$$
$$C[\mathsf{let\ right}\, E\, \mathsf{be\ left}\, x.\, E_1' \mid \mathsf{right}\, y.\, E_2'] \rhd C[E_2'\{y \mapsto E\}]$$

Figure 2.24. Adding sums to the λ-calculus

and their disjunction is

$$\mathsf{let}\ x\ \mathsf{be\ left}\, u.\, y \mid \mathsf{right}\, u.\, \mathsf{right}\langle\rangle. \tag{2.45}$$

Figure 2.24 formalizes the rules for sum types, which correspond to disjunction formulas via the formulas-as-types correspondence. Types T are now generated by the context-free grammar

$$T ::= A \mid T \to T \mid T \times T \mid 1 \mid T + T, \tag{2.46}$$

with a new case $T + T$ at the end. We could also add an empty type, which would correspond to the false formula in logic.

2.3.5. Recursion. If an integer like 2 is an expression of type int, then we can use products to express a pair of integers (such as $\langle 2, 2\rangle$, of type int $\times$ int), as well as chain products to express longer tuples like a triple of integers (such as $\langle 2, \langle 2, 2\rangle\rangle$, of type int $\times$ (int $\times$ int)). To allow for a tuple of 0, 1, 2, or 3 integers, we can use the sum type

$$1 + (\mathrm{int} + ((\mathrm{int} \times \mathrm{int}) + (\mathrm{int} \times (\mathrm{int} \times \mathrm{int})))), \tag{2.47}$$

in which we can express a pair of integers (right(right(left$\langle 2, 2\rangle$))) as well as a triple of integers (right(right(right(left$\langle 2, \langle 2, 2\rangle\rangle$))). Alternatively, we can use the

more concise sum type

$$1 + (\mathrm{int} \times (1 + (\mathrm{int} \times (1 + \mathrm{int})))), \tag{2.48}$$

in which again we can express a pair of integers ($\mathsf{right}\langle 2, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle$) as well as a triple of integers ($\mathsf{right}\langle 2, \mathsf{right}\langle 2, \mathsf{right}\, 2\rangle\rangle$). This pattern of types lets us deal with integer lists of any length, as long as a maximum length is specified. For example, the type (following the pattern in (2.48))

$$1 + (\mathrm{int} \times (1 + (\mathrm{int} \times (1 + (\mathrm{int} \times (1 + (\mathrm{int} \times (1 + (\mathrm{int} \times 1))))))))) \tag{2.49}$$

accommodates up to 5 integers. If y is of this type, then the program

$$\begin{aligned}
&\mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\, u.\, 0 \mid \mathsf{right}\, v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\, u.\, 0 \mid \mathsf{right}\, v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\, u.\, 0 \mid \mathsf{right}\, v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\, u.\, 0 \mid \mathsf{right}\, v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\, u.\, 0 \mid \mathsf{right}\, v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x))))
\end{aligned} \tag{2.50}$$

computes the sum of the integers in the list.

For programs in the real world as well as examples in this dissertation, we seldom want to impose a maximum length limit on lists. Products and sums are not enough to encode lists of arbitrary length. Roughly, we need an "infinite type" like

$$1 + (\mathrm{int} \times (1 + (\mathrm{int} \times (1 + (\mathrm{int} \times (1 + (\mathrm{int} \times (1 + (\mathrm{int} \times \cdots))))))))), \tag{2.51}$$

where "$\cdots$" is a "miniature copy" of the whole type. In general, for any type T (such as int), we can define the type list T by the recursive equation

$$\mathrm{list}\, T = 1 + (T \times \mathrm{list}\, T). \tag{2.52}$$

Informally speaking, (2.52) defines the type of a list whose elements are of type T, by specifying that such a list is either empty or nonempty. If the list is empty, we represent it by the unit type 1. If the list is nonempty, we represent it by a product type $T \times \mathrm{list}\, T$, where the first component T represents the first element of the list, and the second component list T represents the remainder of the list (which in turn can be empty or nonempty, and so on). Because these are the only two alternatives for a list, (2.52) defines list T to be a sum type that recursively refers to list T itself. The type list T is hence called a *recursive type*.

For the sake of presenting example programs below, we informally introduce recursive types and programs here. Pierce (2002) and Gapeyev et al. (2002) give more formal introductions. In particular, this dissertation uses only *inductive types*, which we can add to our programming language while preserving the termination property (Greiner 1992).

A second example of a recursive type is the type tree T of binary trees, defined by

$$\text{tree}\, T = T + (\text{tree}\, T \times \text{tree}\, T), \tag{2.53}$$

where T is the type of leaf labels. This recursive equation defines tree T to be a sum type: a binary tree is either a leaf node (containing some data of type T) or a branch (containing an ordered pair of type tree $T \times$ tree T, recursively). For instance,

$$\mathsf{right}\langle \mathsf{left}\, 1, \mathsf{right}\langle \mathsf{left}\, 2, \mathsf{left}\, 3\rangle\rangle \tag{2.54}$$

is a binary tree, of type tree int. We can think of tree T as the "infinite type"

$$T + ((T + (\cdots \times \cdots)) \times (T + (\cdots \times \cdots))), \tag{2.55}$$

where each "$\cdots$" is a "miniature copy" of the whole type. This "infinite type" is produced by *unrolling* the recursion in (2.53)—that is, by repeatedly substituting (2.53) into itself. Another way to encode binary trees is to define tree T by

$$\text{tree}\, T = T + ((1 + 1) \to \text{tree}\, T). \tag{2.56}$$

The tree in (2.54) is then encoded as

$$\begin{aligned}&\mathsf{right}(\lambda x.\, \mathsf{let}\ x\ \mathsf{be}\ \mathsf{left}\, u.\, \mathsf{left}\, 1\\ &\qquad \mid \mathsf{right}\, u.\, \mathsf{right}(\lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\, u.\, \mathsf{left}\, 2 \mid \mathsf{right}\, u.\, \mathsf{left}\, 3)).\end{aligned} \tag{2.57}$$

As a third example, we can encode integers using a recursive type. An integer is either 0, a positive integer (in other words, a natural number), or a negative integer (in other words, the negation of a natural number). We write nat for the type of natural numbers, defined by the recursive equation

$$\text{nat} = 1 + \text{nat} \tag{2.58}$$

because a natural number is either 1 or the successor of a natural number. Unrolling the recursion gives the "infinite type"

$$1 + (1 + (1 + (1 + \cdots))), \tag{2.59}$$

where "$\cdots$" is a "miniature copy" of the whole type. Having defined nat, we can then define the type int of integers by the (nonrecursive) equation

$$\text{int} = 1 + (\text{nat} + \text{nat}). \tag{2.60}$$

Here 1 is the type of integers that are zero, the first nat is the type of integers that are positive, and the second nat is the type of integers that are negative. For example, we encode the integer 3 as $\mathsf{right}(\mathsf{left}(\mathsf{right}(\mathsf{right}(\mathsf{left}\langle\rangle))))$, where the $\mathsf{right}(\mathsf{left}(\ldots))$ outside means a positive integer, and the $\mathsf{right}(\mathsf{right}(\mathsf{left}\langle\rangle))$ inside means the natural number 3.

Some operations on recursive types can be expressed by ordinary programs. For example, if E is of type $\mathsf{list}\,\mathsf{int}$, then the program (2.50) on page 42 computes the sum of the first 5 integers in the list (or of the entire list, if it is shorter than 5 elements). Other operations on recursive types call for *recursive programs*. Like a recursive type, a recursive program is defined by a recursive equation, and can be thought of as the "infinite program" that results from unrolling the equation. For example, the recursive equation

(2.61) $$F = \lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\, 0 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + Fy$$

defines an "infinite program" F, so that Fy computes the sum of the integers in the list y. This "infinite program" (cf. (2.50))

(2.62) $$\begin{aligned}
&\lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\, 0 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\, 0 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\, 0 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\, 0 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + (\\
&\lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\, 0 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x + \ldots y)y)y)y)y
\end{aligned}$$

is the result of unrolling (2.61). Here "..." is a "miniature copy" of the whole program.

2.4. Type-logical grammar

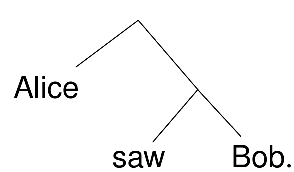
We now turn back to natural language. The combinatory structure of the λ-calculus turns out to be extremely useful for modeling how utterances combine with each other. For the basic intuition, consider a sentence like

(2.63) Alice saw Bob.

We want to model how the pronunciations and meanings of the individual words Alice, saw, and Bob combine to form the pronunciation and meaning of the complete sentence (2.63).

Syntactically, we treat a complex expression as a binary tree whose leaves are atomic expressions. For example, we treat (2.63) as the binary tree

(2.64)



(We revisit this grouping below.) The sequence of leaves in a tree from left to right is the tree's *fringe* (also *frontier* or *yield*). For example, the fringe of (2.64) is Alice saw Bob. If a grammar proves a tree well-formed (in other words, classifies, *admits*, or *generates* the tree), then we also say that the grammar proves the tree's fringe well-formed.

Semantically, we treat the verb saw as a function that takes two arguments, Alice and Bob, to give the sentence (2.63). A simple (or simplistic) theory of meaning goes as follows: Alice denotes the individual Alice; Bob denotes the individual Bob; and saw denotes a relation among individuals, or equivalently, a function from individuals to functions from individuals to Boolean values (true or false). If Alice did see Bob, then the meaning of saw would be a function that maps Bob to a function that maps Alice to true. (Note the argument order: the meaning of saw first takes Bob as argument, then Alice.) A more sophisticated theory of meaning might involve mental concepts, physical references, or functions from possible worlds as in Section 1.5. For example, a sentence might denote a propositional formula, a situation description, or a function from possible worlds to Boolean values. But the basic idea of applying the meaning of saw as a function to the meanings of Bob and Alice as arguments is the same. Whatever meanings are, let us say that Alice means a, Bob means b, and saw means f, so that Alice saw Bob denotes fba.

To express the syntactic assertion that Alice is a noun phrase alongside the semantic assertion that Alice means a, we write the λ-calculus judgment

$$a : \mathsf{Alice} \vdash a : np, \tag{2.65}$$

pronounced "Alice with the meaning a is an np with the meaning a". We set a above in Roman type because it is not in the syntax of the language under discussion, namely English. In the same judgment, Alice and np are both (base) types. That is because Alice and np are both (atomic) classifications of expressions: an expression belongs to the type Alice just in case it is the word Alice, whereas an expression belongs to the type np just in case it is a noun phrase.

As promised in Section 2.1, this representation of expressions tightly couples semantics to syntax: each colon connects the semantic meaning of an expression (to the left) to its syntactic classification (to the right). For example, the judgment (2.65) should feel natural because the antecedent $a : \mathsf{Alice}$ couples the meaning a to the classification Alice, whereas $a : np$ couples the same meaning a to the less informative classification np. There is no way to represent an expression that is well formed but meaningless or ill formed yet meaningful. The turnstile $\vdash$ models entailment among linguistic classifications, so this judgment says that the meaning a of every Alice-expression is the meaning of some np-expression. As in Section 2.3.1, we can take each type to denote a set: we can take np to denote the set of all individuals, and Alice to denote the singleton set of the individual Alice.

Analogously to (2.65), we write

$$b : \mathsf{Bob} \vdash b : np \tag{2.66}$$

to model the word Bob.

$$\dfrac{\dfrac{\dfrac{f : \mathsf{saw} \vdash f : np \to np \to s \quad b : \mathsf{Bob} \vdash b : np}{f : \mathsf{saw}, b : \mathsf{Bob} \vdash fb : np \to s}\to\mathrm{E} \quad a : \mathsf{Alice} \vdash a : np}{(f : \mathsf{saw}, b : \mathsf{Bob}), a : \mathsf{Alice} \vdash fba : s}\to\mathrm{E}}{a : \mathsf{Alice}, (f : \mathsf{saw}, b : \mathsf{Bob}) \vdash fba : s}\,\text{Exchange}$$

(a) Alice saw Bob

$$\dfrac{\begin{array}{c}\vdots\,(a)\\ a : \mathsf{Alice}, (f : \mathsf{saw}, b : \mathsf{Bob}) \vdash fba : s\end{array}}{a : \mathsf{Alice}, ((f : \mathsf{saw}, b : \mathsf{Bob}), c : \mathsf{Carol}) \vdash fba : s}\,\text{Weaken}$$

(b) *Alice saw Bob Carol

$$\dfrac{\dfrac{\dfrac{\dfrac{f : \mathsf{saw} \vdash f : np \to np \to s \quad a : \mathsf{Alice} \vdash a : np}{f : \mathsf{saw}, a : \mathsf{Alice} \vdash fa : np \to s}\to\mathrm{E} \quad a : \mathsf{Alice} \vdash a : np}{(f : \mathsf{saw}, a : \mathsf{Alice}), a : \mathsf{Alice} \vdash faa : s}\to\mathrm{E}}{f : \mathsf{saw}, (a : \mathsf{Alice}, a : \mathsf{Alice}) \vdash faa : s}\,\text{Associate}}{f : \mathsf{saw}, a : \mathsf{Alice} \vdash faa : s}\,\text{Contract}$$

(c) *saw Alice

Figure 2.25. Natural-language derivations in the simply-typed λ-calculus, using structural rules to generate acceptable as well as unacceptable sentences

The verb saw is a bit more complicated. Syntactically, it combines with two noun phrases (np) to form a complete sentence (s, or clause). Semantically, it applies to two noun-phrase meanings (perhaps individuals) to form a complete-sentence meaning (perhaps a Boolean). We write

$$f : \mathsf{saw} \vdash f : np \to np \to s \tag{2.67}$$

to capture and couple these assertions (Ajdukiewicz 1935). Here saw is a base type that classifies just the word saw, whereas $np \to np \to s$ is a function type that classifies those expressions that combine with an np to form an expression that combines with an np to form an s. This judgment says that the meaning f of every expression of type saw is the meaning of some expression of type $np \to np \to s$. As in Section 2.3, we can take each function type to denote a set of functions: the type $np \to np \to s$ denotes the set of functions from individuals to functions from individuals to sentence meanings.

We can now combine these words into a sentence. Figure 2.25a derives Alice

saw Bob (2.63) in the simply-typed λ-calculus (Figure 2.17). As promised at the beginning of this chapter, each word in the sentence enters the proof as a premise. Pairing models juxtaposition of utterances. For example, the judgment

$$f : \mathsf{saw}, b : \mathsf{Bob} \vdash fb : np \to s \tag{2.68}$$

in the proof says that, if an expression is constituted of one part that can be classified as saw and means f, followed by another part that can be classified as Bob and means b, then it can be classified as $np \to s$ and means the function application fb. The formula $np \to s$ here indicates that saw Bob is incomplete; it must be applied to another np (in this case Alice) to form a complete sentence. The final conclusion of the proof says not just that Alice saw Bob means fba, but with a particular ordering (Alice before saw before Bob) and grouping (saw with Bob rather than Alice) of words.

As a model of the structure and meaning of utterances, the simply-typed λ-calculus suffers from several obvious inaccuracies.

- Phrases cannot be reordered freely in natural language, yet that is precisely what the Exchange rule allows. One step before Figure 2.25a concludes correctly that Alice saw Bob is well formed with some meaning, it concludes incorrectly that *saw Bob Alice is also well formed with the same meaning.
- Irrelevant phrases cannot be inserted in natural language, yet that is precisely what the Weaken rule allows. Figure 2.25b concludes incorrectly that *Alice saw Bob Carol is well formed with the same meaning as Alice saw Bob.
- Duplicate phrases cannot be merged in natural language, yet that is precisely what the Contract rule allows. Figure 2.25c concludes incorrectly that *saw Alice is a well-formed sentence that means that Alice saw herself.
- Phrases cannot be regrouped (that is, the binary tree of constituency reparenthesized or rebracketed) freely in natural language, yet that is precisely what the Associate rule allows. Much empirical evidence indicates that grouping matters in natural language. To take an example from English: Without grouping, the string 's mother saw Bob would have the type $np \to s$ just as the verb phrases saw Bob and heard Carol do. Yet, whereas the sentence Alice saw Bob and heard Carol means that Alice saw Bob and Alice heard Carol, the sentence Alice's mother saw Bob and heard Carol does not mean that Alice's mother saw Bob and Alice heard Carol. We follow standard linguistics practice here in grouping a transitive verb like saw more tightly with its object Bob than with its subject Alice.

$$\frac{}{x : T \vdash x : T}\,\mathrm{Id} \qquad \frac{\Gamma,\, x : T_1 \vdash E : T_2}{\Gamma \vdash \lambda x.\, E : T_2/T_1}\,/\,\mathrm{I} \qquad \frac{x : T_1,\, \Gamma \vdash E : T_2}{\Gamma \vdash \lambda x.\, E : T_1\backslash T_2}\,\backslash\,\mathrm{I}$$

$$\frac{\Gamma \vdash F : T_2/T_1 \quad \Delta \vdash E : T_1}{\Gamma, \Delta \vdash FE : T_2}\,/\,\mathrm{E} \qquad \frac{\Delta \vdash E : T_1 \quad \Gamma \vdash F : T_1\backslash T_2}{\Delta, \Gamma \vdash FE : T_2}\,\backslash\,\mathrm{E}$$

Figure 2.26. The non-associative Lambek calculus without products, with semantics

To deal with these problems, we remove all four structural rules (Weaken, Contract, Exchange, Associate) from the grammar. However, Figure 2.25a uses Exchange to derive correctly that Alice saw Bob is acceptable. This use of Exchange is crucial, because the $\rightarrow$E rule in Figure 2.17 only allows a function to take an argument to its right, yet saw Bob needs to take its argument Alice to its left. Without Exchange, we need some other way to apply a function to an argument to the left.

Lambek (1958), following Bar-Hillel (1953), solves the problem by splitting functions into two groups: those that take arguments to the right (of type T_2/T_1, pronounced "T_2 from T_1" or "T_2 over T_1") and those that take arguments to the left (of type $T_1\backslash T_2$, pronounced "T_1 into T_2" or "T_1 under T_2"). Figure 2.26 shows the result of the split, with all structural rules removed. Each of the two function connectives, / and \, has its own introduction and elimination rules. In this new system, we can assign saw the type $(np\backslash s)/np$ and represent Alice saw Bob (2.63) by the following proof.

$$\frac{a : \mathsf{Alice} \vdash a : np \qquad \dfrac{f : \mathsf{saw} \vdash f : (np\backslash s)/np \quad b : \mathsf{Bob} \vdash b : np}{f : \mathsf{saw},\, b : \mathsf{Bob} \vdash fb : np\backslash s}\,/\,\mathrm{E}}{a : \mathsf{Alice},\, (f : \mathsf{saw},\, b : \mathsf{Bob}) \vdash fba : s}\,\backslash\,\mathrm{E} \tag{2.69}$$

One way to understand the inference rules in Figure 2.26 is to think of each word or phrase as a conserved resource. For example, the verb saw has the type $(np\backslash s)/np$ because it first consumes an np to its right, then consumes an np to its left, finally to give an s. Without structural rules, resources can no longer be freely discarded, duplicated, reordered, or regrouped. Hence this is a *substructural* logic (Restall 2000), more specifically *multiplicative linear logic* (Girard 1987) with neither commutativity (the Exchange rule) nor associativity (the Associate rule).

By convention, the slashes / and \ associate towards the result type. For example, $c/b/a$ means the type $(c/b)/a$, whereas $a\backslash b\backslash c$ means the type $a\backslash(b\backslash c)$.

Figure 2.26 is called the *non-associative Lambek calculus without products*. It is an instance of *type-logical grammar* (or *categorial type logic*) in that it uses connectives like / and \ in types to classify utterances and govern their

$$\frac{}{T \vdash T}\,\mathrm{Id} \qquad \frac{\Gamma, T_1 \vdash T_2}{\Gamma \vdash T_2/T_1}\,/\,\mathrm{I} \qquad \frac{T_1, \Gamma \vdash T_2}{\Gamma \vdash T_1\backslash T_2}\,\backslash\,\mathrm{I}$$

$$\frac{\Gamma \vdash T_2/T_1 \quad \Delta \vdash T_1}{\Gamma, \Delta \vdash T_2}\,/\,\mathrm{E} \qquad \frac{\Delta \vdash T_1 \quad \Gamma \vdash T_1\backslash T_2}{\Delta, \Gamma \vdash T_2}\,\backslash\,\mathrm{E}$$

Figure 2.27. The non-associative Lambek calculus without products, without semantics

combination in a substructural logic (Moortgat 1997). As noted earlier, one distinguishing property of type-logical grammar is its tight coupling of syntax and semantics: Via the formulas-as-types correspondence, the proof that classifies an utterance as well-formed dictates the utterance's meaning. In particular, the / I and \ I rules build functions, whereas / E and \ E apply functions. This coupling is so tight that, for brevity, we omit meanings in most derivations below. Figure 2.27 shows the same inference rules so abbreviated. This abbreviation should not distract from the crucial constraints that semantics places on linguistic theory. As we turn below to additional linguistic phenomena like quantification, we often decide how to analyze them by considering what phrases can mean.

2.4.1. Multimodal type-logical grammar. Section 2.1 above gives three reasons why context-free grammars are not ideal for modeling natural language.

- They do not exist for some languages.
- They are awkward for some linguistic phenomena.
- They enforce no correspondence between syntax and semantics.

So far we have only addressed the last point, using the formulas-as-types correspondence. As for the first two points (in short, expressiveness), we actually still cannot describe any set of strings that is not a context-free language: Context-free grammars and non-associative Lambek grammars generate exactly the same string-sets. In other words, they have the same weak generative capacity (Buszkowski 1986; see also Kandulski 1988 for the Lambek calculus with products, and Pentus 1993, 1997, 1996 for variations with associativity).

To go beyond context-free languages, we introduce additional *modes* of combination besides juxtaposition (Moortgat 1997; Section 4 and references therein). Most intuitive understanding for these modes comes from their use, which abounds in Chapter 4. In this section, we just describe the machinery and demonstrate it with a formal language alluded to in Section 2.1: the set of doubled strings $\{ ww \mid w \in \Sigma^* \}$, which is not context-free.

The basic idea behind *multimodal* type-logical grammar is that utterances may combine in more than one way, and some of these ways may not have arity

two. For intuition, imagine that we are modeling not utterances pronounced over time but visual elements laid out on a page (Henderson 1982) or musical notes arranged in a score (Hudak et al. 1996). Two visual elements may be combined by juxtaposing them horizontally or vertically (two binary modes of combination); one visual element may be embellished by framing it with a box (a unary mode of combination). Two musical notes may be combined by playing them at the same time or one after another (two binary modes of combination); one musical note may be embellished with dynamics like forte (a unary mode of combination).

While it may be less obvious at first glance, it is also useful to posit multiple ways to combine utterances in spoken language: juxtaposition alone is often not enough to encode the desired syntax and semantics. Indeed, our linguistic analyses in Chapter 4 rely crucially on adding binary and unary modes and specifying how they interact.

For example, to analyze the copular sentence

(2.70) Alice is a woman,

it is intuitive to take the noun woman to mean a function that applies to the meaning of Alice to yield the meaning of the sentence: perhaps woman is a function from individuals to Booleans that maps women to true and others to false. In other words, woman is a noun that predicates of Alice. However, we cannot assign the type $np\backslash s$ or s/np to woman, because neither *Alice woman nor *woman Alice is an acceptable sentence in English. Rather, Section 4.3.1 below introduces a new binary mode $,_n$ (the subscript n being mnemonic for "noun") for combining something with a noun that predicates of it. We then assign the type $np\backslash_n s$ to woman, so that the judgment

(2.71) $\text{Alice},_n \text{woman} \vdash s$

is derivable, while the judgment

(2.72) $\text{Alice}, \text{woman} \vdash s$

remains not derivable.

To take another example, the wh-question who Alice saw in the sentence

(2.73) Bob knows who Alice saw

can be analyzed as the function $\lambda x.\, fxa$, where f and a are the meanings of saw and Alice as above. In general, we can analyze a wh-question to mean a function from a short answer (such as Carol) to a proposition (such as that Alice saw Carol) (Krifka 2001). However, who Alice saw cannot have the type $np\backslash s$ or s/np, because neither *Carol who Alice saw nor *who Alice saw Carol is an

$$\frac{\Gamma ,_m x : T_1 \vdash E : T_2}{\Gamma \vdash \lambda x.\, E : T_2 /_m T_1}\, /_m \mathrm{I} \qquad \frac{x : T_1 ,_m \Gamma \vdash E : T_2}{\Gamma \vdash \lambda x.\, E : T_1 \backslash_m T_2}\, \backslash_m \mathrm{I}$$

$$\frac{\Gamma \vdash F : T_2 /_m T_1 \quad \Delta \vdash E : T_1}{\Gamma ,_m \Delta \vdash FE : T_2}\, /_m \mathrm{E} \qquad \frac{\Delta \vdash E : T_1 \quad \Gamma \vdash F : T_1 \backslash_m T_2}{\Delta ,_m \Gamma \vdash FE : T_2}\, \backslash_m \mathrm{E}$$

Figure 2.28. Adding a binary mode m to the Lambek calculus without products

acceptable sentence in English, let alone one that means that Alice saw Carol. Rather, Section 4.5.2 below introduces a new binary mode $,_?$ for combining something with a question that predicates of it. We then arrange for our grammar to derive the type $np\backslash_? s$ for who Alice saw, so that the judgment

$$\mathsf{Carol},_? (\mathsf{who}, (\mathsf{Alice}, \mathsf{saw})) \vdash s \tag{2.74}$$

is derivable, while the judgment

$$\mathsf{Carol}, (\mathsf{who}, (\mathsf{Alice}, \mathsf{saw})) \vdash s \tag{2.75}$$

remains not derivable.

For the linguistic applications in Chapter 4, yet another binary mode is crucial. In Section 4.1, we treat a context as an expression in its own right. We then introduce the *continuation mode*, which combines a subexpression with a context to form a larger expression by plugging the subexpression into the context, or equivalently, by wrapping the context around the subexpression. Informally speaking, we arrange for the continuation mode to combine the subexpression Bob with the context Alice saw [ ]'s mother to form the larger expression Alice saw Bob's mother.

Motivated by these examples, we now formalize multiple modes of combination. The non-associative Lambek calculus (without products) in Figures 2.26 and 2.27 (on pages 48–49) has a single, binary mode of combination, which formally models the juxtaposition of expressions with grouping. Thus the syntax of an expression is treated as a binary tree with a single kind of branch nodes, unlabeled in (2.64) on page 44. This mode of combination is called the *default mode*. In an antecedent, it is represented by the comma.

To add a new binary mode to type-logical grammar, we simply duplicate the comma and the connectives / and \, as follows. We first pick a letter, say m, to distinguish this new mode from others. We let two type environments Γ and Δ be combined using a new comma "$,_m$" into a larger type environment $\Gamma ,_m \Delta$. We also add types of the form $T_2 /_m T_1$ and $T_1 \backslash_m T_2$, where T_1 and T_2 are types. Finally, we add the inference rules in Figure 2.28. These rules are just the rules in Figure 2.26 (except Id) with m subscripted everywhere. Just as the semantics in

$$\frac{\langle \Gamma \rangle_m \vdash E : T}{\Gamma \vdash E : \Box^{\downarrow}_m T}\,\Box^{\downarrow}_m \mathrm{I} \qquad \frac{\Gamma \vdash E : T}{\langle \Gamma \rangle_m \vdash E : \Diamond_m T}\,\Diamond_m \mathrm{I}$$

$$\frac{\Gamma \vdash E : \Box^{\downarrow}_m T}{\langle \Gamma \rangle_m \vdash E : T}\,\Box^{\downarrow}_m \mathrm{E} \qquad \frac{\Delta \vdash E : \Diamond_m T_0 \quad \Gamma[\langle x : T_0 \rangle_m] \vdash E' : T}{\Gamma[\Delta] \vdash \text{let } E \text{ be } x.\, E' : T}\,\Diamond_m \mathrm{E}$$

Figure 2.29. Adding a unary mode m to the Lambek calculus

Figure 2.26 does not distinguish between / and \ in abstractions and applications, the semantics here does not distinguish $/_m$, $\backslash_m$, /, and \ from each other. This choice reflects the popular intuition that different binary modes regulate just syntactic combination.

To add a unary mode to type-logical grammar, we again pick a letter m. We then let a type environment Γ be enclosed in a pair of angle brackets $\langle\ \rangle_m$ to form a larger type environment $\langle \Gamma \rangle_m$. We also add types of the form $\Diamond_m T$ and $\Box^{\downarrow}_m T$, where T is a type. Finally, we add the inference rules in Figure 2.29. The unary type connective $\Diamond_m$ and the corresponding unary structural punctuation $\langle\ \rangle_m$ represent this new mode of combination in types and type environments, much as the binary structural punctuation , (comma) represents juxtaposition in type environments. By convention, unary connectives have higher precedence than binary connectives, so $\Diamond_m a/b$ means the type $(\Diamond_m a)/b$ rather than $\Diamond_m(a/b)$.

The category theorist will recognize that these rules specify that $\Diamond_m$ and $\Box^{\downarrow}_m$ are a pair of functors that form an adjunction. The modal logician can take $\Diamond_m$ to mean "in some world accessible from here" and take $\Box^{\downarrow}_m$ to mean "in every world accessible to here". The reader who is more comfortable with binary modes can treat $\langle \Gamma \rangle_m$ as $t,_m \Gamma$, $\Diamond_m T$ as $t,_m T$, and $\Box^{\downarrow}_m T$ as $t\backslash_m T$, for some base type t and binary mode m. The rules for the unary m mode then follow from Id and the rules for the binary m mode (including for products).

The semantics shown in Figure 2.29 essentially forgets the unary mode; for example, the premise and conclusion have the same meaning in the first three rules. This choice reflects how the linguistic analyses in this dissertation use unary modes, namely to regulate just syntactic combination as we now demonstrate briefly. Moortgat (1997; Section 4.2.1) describes another semantics for unary modes that attributes semantic significance to them, such as intensionality and information structure.

A multimodal type-logical grammar can contain any number of modes, of any arity (Dunn 1991), though we only need binary and unary modes. Each mode can have its own structural rules. For example, if we want a binary mode m to be associative but not the default mode, we can leave out the Associate rule for the default mode in Figure 2.17, as discussed above, but add the corresponding rule

for mode m:

$$\frac{\Gamma[(\Delta ,_m \Theta) ,_m \Pi] \vdash E : T}{\Gamma[\Delta ,_m (\Theta ,_m \Pi)] \vdash E : T}\ \text{Associate}_m. \tag{2.76}$$

Moreover, modes can interact through *mixed* structural rules, which are structural rules that mention multiple modes. For example, if we want to allow reordering certain phrases, we can introduce a unary mode m, and arrange for reorderable phrases to have types of the form $\Diamond_m T$. The default (binary) mode and the new unary mode m can then communicate through the following Exchange-like mixed rule.

$$\frac{\Gamma[\langle \Delta \rangle_m , \langle \Theta \rangle_m] \vdash E : T}{\Gamma[\langle \Theta \rangle_m , \langle \Delta \rangle_m] \vdash E : T} \tag{2.77}$$

This rule allows a pair of juxtaposed phrases to be reordered as long as their types are both enclosed by $\Diamond_m$.

With multiple modes of combination, we need to redefine what it means for a grammar to generate a string. A multimodal grammar designates certain modes, including the default mode, as *external*. We say that the grammar generates a string if the grammar proves a tree well-formed whose fringe is the string, and which is built up from atomic expressions using only external modes. For example, because we do not designate the binary modes n and ? as external in Chapter 4, deriving the judgments (2.71) and (2.74) on pages 50–51 does not generate the strings *Alice woman and *Carol who Alice saw (that is, does not predict the strings to be acceptable sentences). Informally speaking, a mode is external just in case a speaker can pronounce it. Otherwise, it is *internal*.

To demonstrate all this machinery, Figure 2.30 shows a multimodal grammar that generates the context-sensitive language of strings of the form ww, where w is any string over the alphabet $\{\mathsf{a}, \mathsf{b}\}$. This grammar uses three modes: the default binary mode (unnamed), a new binary mode d, and a new unary mode (unnamed). Only the default mode is external.

The idea behind this grammar is to create structures $\Delta ,_d \Theta$ such that Δ and Θ are identical. To achieve this, the lexical entries create and juxtapose copies of the structures $\mathsf{a} ,_d \mathsf{a}$ and $\mathsf{b} ,_d \mathsf{b}$. Structural rule 2 then lets the d mode "bubble up" to the top level. Figure 2.30 derives the doubled string aabaab using structural rule 2 twice. Finally (at the bottom of the derivation), structural rule 1 changes the top-level composition mode from d to default, which renders the antecedent pronounceable. Because the d mode only combines a with a and b with b in the lexicon, this grammar generates only strings of the form ww.

Lexicon

$$\mathsf{a} \vdash \Box^{\downarrow} s /_d \mathsf{a} \qquad \mathsf{a} \vdash (\Box^{\downarrow} s \backslash \Box^{\downarrow} s) /_d \mathsf{a} \qquad \mathsf{b} \vdash \Box^{\downarrow} s /_d \mathsf{b} \qquad \mathsf{b} \vdash (\Box^{\downarrow} s \backslash \Box^{\downarrow} s) /_d \mathsf{b}$$

Structural rules

$$\frac{\Gamma[\langle \Delta ,_d \Theta \rangle] \vdash E : T}{\Gamma[\Delta, \Theta] \vdash E : T}\, 1 \qquad \frac{\Gamma[(\Delta_1 ,_d \Theta_1), (\Delta_2 ,_d \Theta_2)] \vdash E : T}{\Gamma[(\Delta_1, \Delta_2) ,_d (\Theta_1, \Theta_2)] \vdash E : T}\, 2$$

Sample derivation of aabaab

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{a} \vdash \Box^{\downarrow} s /_d \mathsf{a} \quad \dfrac{}{\mathsf{a} \vdash \mathsf{a}}\,\mathrm{Id}}{\mathsf{a} ,_d \mathsf{a} \vdash \Box^{\downarrow} s}\, /_d \mathrm{E} \quad \dfrac{\mathsf{a} \vdash (\Box^{\downarrow} s \backslash \Box^{\downarrow} s) /_d \mathsf{a} \quad \dfrac{}{\mathsf{a} \vdash \mathsf{a}}\,\mathrm{Id}}{\mathsf{a} ,_d \mathsf{a} \vdash \Box^{\downarrow} s \backslash \Box^{\downarrow} s}\, /_d \mathrm{E}}{(\mathsf{a} ,_d \mathsf{a}), (\mathsf{a} ,_d \mathsf{a}) \vdash \Box^{\downarrow} s}\, \backslash \mathrm{E} \quad \dfrac{\mathsf{b} \vdash (\Box^{\downarrow} s \backslash \Box^{\downarrow} s) /_d \mathsf{b} \quad \dfrac{}{\mathsf{b} \vdash \mathsf{b}}\,\mathrm{Id}}{\mathsf{b} ,_d \mathsf{b} \vdash \Box^{\downarrow} s \backslash \Box^{\downarrow} s}\, /_d \mathrm{E}}{((\mathsf{a} ,_d \mathsf{a}), (\mathsf{a} ,_d \mathsf{a})), (\mathsf{b} ,_d \mathsf{b}) \vdash \Box^{\downarrow} s}\, \backslash \mathrm{E}}{((\mathsf{a}, \mathsf{a}) ,_d (\mathsf{a}, \mathsf{a})), (\mathsf{b} ,_d \mathsf{b}) \vdash \Box^{\downarrow} s}\, 2}{((\mathsf{a}, \mathsf{a}), \mathsf{b}) ,_d ((\mathsf{a}, \mathsf{a}), \mathsf{b}) \vdash \Box^{\downarrow} s}\, 2}{\langle ((\mathsf{a}, \mathsf{a}), \mathsf{b}) ,_d ((\mathsf{a}, \mathsf{a}), \mathsf{b}) \rangle \vdash s}\, \Box^{\downarrow} \mathrm{E}}{((\mathsf{a}, \mathsf{a}), \mathsf{b}), ((\mathsf{a}, \mathsf{a}), \mathsf{b}) \vdash s}\, 1$$

Figure 2.30. A multimodal type-logical grammar that generates the context-sensitive language of doubled strings $\{ ww \mid w \in \{\mathsf{a}, \mathsf{b}\}^* \}$

CHAPTER 3

The analogy: delimited control and quantification

In Chapter 1, we saw two instances of similarity between a computational side effect and a linguistic side effect: state can be analyzed like anaphora, and environment can be analyzed like intensionality. Armed with the formal tools from Chapter 2, we now consider a third instance: *delimited control* can be analyzed like *quantification*. As in Chapter 1, we first present the two side effects and how they have been treated in programming-language theory and linguistics. We then make the new observation that the side effects are similar, so as to come up with better treatments in Chapters 4 and 5.

3.1. Delimited control

The model of computation embodied by the λ-calculus of Section 2.3 is "hierarchical": two disjoint subexpressions of the same program execute in isolation. For example, the program

$$(\lambda x.\, x)2 + (\lambda y.\, y)3 \tag{3.1}$$

contains two disjoint subexpressions $(\lambda x.\, x)2$ and $(\lambda y.\, y)3$. Either subexpression can undergo β-reduction separately, in either order, without affecting the other or changing the final result 5 of the program.

$$(\lambda x.\, x)2 + (\lambda y.\, y)3 \rhd 2 + (\lambda y.\, y)3 \rhd 2 + 3 \rhd 5 \tag{3.2}$$

$$(\lambda x.\, x)2 + (\lambda y.\, y)3 \rhd (\lambda x.\, x)2 + 3 \rhd 2 + 3 \rhd 5 \tag{3.3}$$

Whenever a context (like $[\,] + (\lambda y.\, y)3$) waits to be plugged with an evaluation result (like 2), only one subexpression (like $(\lambda x.\, x)2$) may evaluate to provide that evaluation result. This hierarchy of computation simplifies parallel execution: should a second processor become available, it can evaluate a chosen subexpression concurrently and provide the result separately without interfering with or being interfered with by the evaluation of the rest of the program, that is, the context of the chosen subexpression.

Sometimes, however, we want multiple ways to plug a context. For example, suppose that we have a list of integers, of type list int from Section 2.3.5, and want to compute the product of its elements. The recursive function (cf. (2.61) on

page 44)

(3.4) $P_1 = \lambda y.\, \mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\, 1 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x \times P_1 y$

does the job. For example, applying P_1 to the list

(3.5) $L_1 = \mathsf{right}\langle 4, \mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\rangle$

gives the program $P_1 L_1$, which executes as follows. (Ellipsis below stands for $\mathsf{left}\ u.\, 1 \mid \mathsf{right}\ v.\, \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\, x \times P_1 y$.)

(3.6) $P_1(\mathsf{right}\langle 4, \mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\rangle)$

$\triangleright\ \mathsf{let}\ \mathsf{right}\langle 4, \mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\rangle\ \mathsf{be}\ \ldots$

$\triangleright\ \mathsf{let}\ \langle 4, \mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\rangle\ \mathsf{be}\ \langle x, y\rangle.\, x \times P_1 y$

$\triangleright\ 4 \times P_1(\mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle)$

$\triangleright\ 4 \times \mathsf{let}\ \mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\ \mathsf{be}\ \ldots$

$\triangleright\ 4 \times \mathsf{let}\ \langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\ \mathsf{be}\ \langle x, y\rangle.\, x \times P_1 y$

$\triangleright\ 4 \times (3 \times P_1(\mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle))$

$\triangleright\ 4 \times (3 \times \mathsf{let}\ \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\ \mathsf{be}\ \ldots)$

$\triangleright\ 4 \times (3 \times \mathsf{let}\ \langle 2, \mathsf{left}\langle\rangle\rangle\ \mathsf{be}\ \langle x, y\rangle.\, x \times P_1 y)$

$\triangleright\ 4 \times (3 \times (2 \times P_1(\mathsf{left}\langle\rangle)))$

$\triangleright\ 4 \times (3 \times (2 \times \mathsf{let}\ \mathsf{left}\langle\rangle\ \mathsf{be}\ \ldots))$

$\triangleright\ 4 \times (3 \times (2 \times 1))$

$\triangleright\ 4 \times (3 \times 2)$

$\triangleright\ 4 \times 6$

$\triangleright\ 24$

However, this solution may compute the product for every suffix of the given list. If we already know that 0 appears in the list, then we also know that the product is 0, so we need not compute the product for any remaining suffix of the list. For example, to compute the product for the list

(3.7) $L_2 = \mathsf{right}\langle 4, \mathsf{right}\langle 0, \mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\rangle\rangle,$

there is no need to calculate 3×2, or even 4×0, once we see 0.

To make it easy to optimize computing the product of a list with 0 in it, we extend our programming language with features for delimited control in Section 3.1.1 below. Despite the familiarity of these features, our initial attempt at these features makes the language dangerously nonconfluent. We then fix the problem in Section 3.1.2 by restricting the computation relation using the notions of values and evaluation contexts.

3.1.1. A first attempt. To implement this optimization, we extend our programming language with two constructs, # (pronounced "prompt" or "reset") and abort. The reader familiar with exception handling in programming languages may think of abort as a primitive variant of throw, and # as a primitive variant of try ... catch.

If E is an expression, then so are $\#E$ and $\mathsf{abort}\ E$. If we were working in the untyped rather than simply-typed λ-calculus, then we would add to Figure 2.14 on page 31 the inference rules

$$\frac{\Gamma \vdash E}{\Gamma \vdash \#E} \tag{3.8}$$

and

$$\frac{\Gamma \vdash E}{\Gamma \vdash \mathsf{abort}\ E}. \tag{3.9}$$

We leave it to Section 3.2.2, after describing the semantics of # and abort, to extend the simply-typed λ-calculus with formal typing rules for these constructs.

With # and abort, we can write the recursive function

$$P_2 = \lambda y.\, \#(P_2' y) \tag{3.10}$$

$$\begin{aligned} P_2' = \lambda y.\, &\mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\ 1 \\ &\mid \mathsf{right}\ v.\ \mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle. \\ &\qquad (\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{abort}\ 0\ \mathsf{else}\ x) \times P_2' y \end{aligned} \tag{3.11}$$

and apply P_2 to the given list. In words, abort 0 means to replace the subexpression inside the innermost enclosing # by 0, that is, answer 0 to the #. By "the subexpression inside the innermost enclosing #", we mean the smallest subexpression $\#E$ that contains abort 0 in the running program when abort 0 takes effect, not in the initial written program.

In a first attempt to formalize how # and abort execute, we could add the following steps to the computation relation.

$$C[\#E] \rhd C[E] \tag{3.12}$$

$$C[\#(D[\mathsf{abort}\ E'])] \rhd C[\#E'] \tag{3.13}$$

The metavariable $D[\ ]$ stands for an *evaluation subcontext*, or *subcontext* for short: an evaluation context whose [ ] is not enclosed in #. This proviso matches abort to the *innermost* enclosing #. Because subcontexts are crucial to delimited control, we formalize the notion in Figure 3.1. This figure defines a new judgment form $D[\ ] : d$, which means that $D[\ ]$ is a subcontext. The two inference rules at the bottom of the figure redefine contexts $C[\ ]$ in terms of subcontexts; these two rules are equivalent to Figures 2.17, 2.22, 2.23, and 2.24 taken together.

Evaluation subcontexts $D[\ ] : d$

$$\frac{}{[\ ] : d} \qquad \frac{D[\ ] : d}{D[\lambda x.\ [\ ]] : d} \qquad \frac{D[\ ] : d \quad \Delta \vdash E : T_1}{D[[\ ]E] : d} \qquad \frac{\Gamma \vdash F : T_0 \quad D[\ ] : d}{D[F[\ ]] : d}$$

$$\frac{D[\ ] : d \quad \Delta \vdash E_2 : T_2}{D[\langle [\ ], E_2 \rangle] : d} \qquad \frac{\Gamma \vdash E_1 : T_1 \quad D[\ ] : d}{D[\langle E_1, [\ ] \rangle] : d} \qquad \frac{D[\ ] : d \quad \Gamma \vdash E' : T'}{D[\mathsf{let}\ [\ ]\ \mathsf{be}\ \langle x, y \rangle.\ E'] : d}$$

$$\frac{\Delta \vdash E : T_0 \quad D[\ ] : d}{D[\mathsf{let}\ E\ \mathsf{be}\ \langle x, y \rangle.\ [\ ]] : d} \qquad \frac{D[\ ] : d \quad \Gamma \vdash E' : T'}{D[\mathsf{let}\ [\ ]\ \mathsf{be}\ \langle \rangle.\ E'] : d} \qquad \frac{\Delta \vdash E : T_0 \quad D[\ ] : d}{D[\mathsf{let}\ E\ \mathsf{be}\ \langle \rangle.\ [\ ]] : d}$$

$$\frac{D[\ ] : d}{D[\mathsf{left}\ [\ ]] : d} \qquad \frac{D[\ ] : d}{D[\mathsf{right}\ [\ ]] : d} \qquad \frac{D[\ ] : d \quad \Gamma_1 \vdash E_1' : T_1' \quad \Gamma_2 \vdash E_2' : T_2'}{D[\mathsf{let}\ [\ ]\ \mathsf{be}\ \mathsf{left}\ x.\ E_1' \mid \mathsf{right}\ y.\ E_2'] : d}$$

$$\frac{\Delta \vdash E : T_0 \quad D[\ ] : d \quad \Gamma \vdash E_2' : T'}{D[\mathsf{let}\ E\ \mathsf{be}\ \mathsf{left}\ x.\ [\ ] \mid \mathsf{right}\ y.\ E_2'] : d} \qquad \frac{\Delta \vdash E : T_0 \quad \Gamma \vdash E_1' : T' \quad D[\ ] : d}{D[\mathsf{let}\ E\ \mathsf{be}\ \mathsf{left}\ x.\ E_1' \mid \mathsf{right}\ y.\ [\ ]] : d}$$

Evaluation contexts $C[\ ]$

$$\frac{D[\ ] : d}{D[\ ]} \qquad \frac{C[\ ]}{C[\#(D[\ ])]}$$

Figure 3.1. Defining evaluation subcontexts and redefining evaluation contexts

The intention of the new constructs # and abort is for the program P_2L_2 to execute as follows. (Ellipsis below stands for $\langle x, y \rangle$. (if x = 0 then abort 0 else x) × $P_2'y$.)

(3.14)

P_2 (right⟨4, right⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩)

▷ #(let right⟨4, right⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩ be left u. 1
| right v. let v be . . .)

▷ #(let ⟨4, right⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩ be . . .)

▷ #((if 4 = 0 then abort 0 else 4) × P_2'(right⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩))

▷ #((if false then abort 0 else 4) × P_2'(right⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩))

▷ #(4 × P_2'(right⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩))

▷ #(4 × let right⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩ be left u. 1 | right v. let v be . . .)

▷ #(4 × let ⟨0, right⟨3, right⟨2, left⟨⟩⟩⟩⟩ be . . .)

▷ #(4 × ((if 0 = 0 then abort 0 else 0) × P_2'(right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩))

▷ #(4 × ((if true then abort 0 else 0) × P_2'(right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩))

▷ #(4 × ((abort 0) × P_2'(right⟨3, right⟨2, left⟨⟩⟩⟩⟩⟩))

$\triangleright$ #0 by (3.13)

$\triangleright$ 0 by (3.12)

The context #[ ] in P_2 can be plugged in two ways—either by sequentially completing the recursion over the entire list, or by abort. The steps above take the second way out before inspecting right⟨3, right⟨2, left⟨⟩⟩⟩ at all, as desired. We say that the *control operator* abort transfers *control* in the program nonlocally, from within P_2' to the *control delimiter* # (Felleisen 1987, 1988). Another way to understand this behavior is that abort actively takes control over, rather than passively reporting to, its evaluation subcontext: the subcontext D[ ] in (3.13) expects to receive an evaluation result from abort E', but is instead discarded by abort E' without receiving anything.

Unfortunately, our computation relation is too permissive for this approach to work as is. Although the desired sequence of computation steps in (3.14) is possible, so are many undesired sequences. First, because

(3.15) $(\lambda y.\,\mathsf{let}\ y\ \mathsf{be}\ \mathsf{left}\ u.\,1$
$\quad \mid \mathsf{right}\ v.\,\mathsf{let}\ v\ \mathsf{be}\ \langle x, y\rangle.\,(\mathsf{if}\ x = 0\ \mathsf{then}\ [\ ]\ \mathsf{else}\ x) \times P_2' y)y$

is an evaluation subcontext, abort 0 in P_2 may take effect right away, before the function is applied to any list y, with or without 0.

(3.16) $$P_2 \triangleright \lambda y.\,\#0 \triangleright \lambda y.\,0$$

If y contains no 0, this premature evaluation of abort then produces the wrong product. Second, according to (3.12) on page 57, the # in P_2 may be removed at any time, not necessarily as late as in (3.14).

(3.17) $$P_2 \triangleright \lambda y.\,P_2' y$$

This premature evaluation of # then prevents abort 0 from matching the # in P_2. Third, because

(3.18) $$\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{abort}\ 0\ \mathsf{else}\ [\ ]$$

is an evaluation context, even when P_2 encounters 0 in the list, it may not skip the rest of the list. For example, an alternative ending to (3.14) is

(3.19) $P_2 L_2 \triangleright^+ \#(4 \times ((\mathsf{abort}\ 0) \times P_2'(\mathsf{right}\langle 3, \mathsf{right}\langle 2, \mathsf{left}\langle\rangle\rangle\rangle\rangle)))$
$\triangleright^+ \#(4 \times ((\mathsf{abort}\ 0) \times 6))$
$\triangleright \#0$ by (3.13)
$\triangleright 0$ by (3.12).

In this scenario, abort 0 takes effect only after the partial product 6 is computed for the rest of the list.

Values $\Gamma \vdash_{\mathrm{V}} V : T$

$$\frac{}{x : T \vdash_{\mathrm{V}} x : T}\ \mathrm{Id}_{\mathrm{V}} \qquad \frac{\Gamma,\, x : T_1 \vdash E : T_2}{\Gamma \vdash_{\mathrm{V}} \lambda x.\, E : T_1 \to T_2} \to \mathrm{I}_{\mathrm{V}} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V_1 : T_1 \quad \Delta \vdash_{\mathrm{V}} V_2 : T_2}{\Gamma, \Delta \vdash_{\mathrm{V}} \langle V_1, V_2 \rangle : T_1 \times T_2} \times \mathrm{I}_{\mathrm{V}}$$

$$\frac{}{\cdot \vdash_{\mathrm{V}} \langle\rangle : 1}\ 1\,\mathrm{I}_{\mathrm{V}} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V : T_1}{\Gamma \vdash_{\mathrm{V}} \mathsf{left}\, V : T_1 + T_2} + \mathrm{I}_{\mathrm{V}} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V : T_2}{\Gamma \vdash_{\mathrm{V}} \mathsf{right}\, V : T_1 + T_2} + \mathrm{I}_{\mathrm{V}}$$

$$\frac{\Gamma[\Delta] \vdash_{\mathrm{V}} V : T}{\Gamma[\Delta, \Theta] \vdash_{\mathrm{V}} V : T}\ \mathrm{Weaken}_{\mathrm{V}} \qquad \frac{\Gamma[\Delta, \Delta] \vdash_{\mathrm{V}} V : T}{\Gamma[\Delta] \vdash_{\mathrm{V}} V : T}\ \mathrm{Contract}_{\mathrm{V}}$$

$$\frac{\Gamma[\Delta, \Theta] \vdash_{\mathrm{V}} V : T}{\Gamma[\Theta, \Delta] \vdash_{\mathrm{V}} V : T}\ \mathrm{Exchange}_{\mathrm{V}} \qquad \frac{\Gamma[(\Delta, \Theta), \Pi] \vdash_{\mathrm{V}} V : T}{\Gamma[\Delta, (\Theta, \Pi)] \vdash_{\mathrm{V}} V : T}\ \mathrm{Associate}_{\mathrm{V}}$$

Figure 3.2. Defining values in the λ-calculus

Without # and abort, our programming language is confluent and strongly normalizing, so every way to execute a given program yields the same outcome. The new constructs give rise to some desirable computation sequences but also some undesirable ones. To distinguish the baby from the bathwater and address the problems illustrated above, we restrict the computation relation to a deterministic one below.

3.1.2. Enforcing call-by-value, left-to-right evaluation. Following Felleisen (1987) in this section, we define a new notion of *values* and modify our existing notion of evaluation contexts, so as to shrink the computation relation for our language to one that is deterministic in that it enforces *call-by-value*, *left-to-right* evaluation.

Figure 3.2 defines certain expressions to be *values*. As in Section 2.2, a value is the result of a successful computation, as opposed to an expression that can still compute to a subsequent expression or that is stuck due to an error. (As mentioned in Section 2.3.1, the last case is impossible for well-formed terms in the simply-typed λ-calculus.) We introduce a new judgment form

$$\Gamma \vdash_{\mathrm{V}} V : T. \tag{3.20}$$

It means that V is a value expression of type T in the type environment Γ. The metavariable V stands for an expression that is a value. It is easy to check that $\Gamma \vdash V : T$ is provable whenever $\Gamma \vdash_{\mathrm{V}} V : T$ is provable.

As the inference rules in Figure 3.2 indicate, a variable or an abstraction is always a value, even if the body of the abstraction is not a value. An expression that introduces a product, unit, or sum type—in other words, an expression of

the form $\langle E_1, E_2\rangle$, $\langle\rangle$, $\mathsf{left}\, E$, or $\mathsf{right}\, E$—is a value just in case its parts E, E_1, E_2 are values. No other expression is a value. For example, $\lambda f.\, fx$, $\langle\rangle$, and $\langle f, \mathsf{left}\langle\rangle\rangle$ are values in an appropriate environment, whereas $\mathsf{let}\ x\ \mathsf{be}\ \langle y, z\rangle.\ \langle y, z\rangle$, $\#\langle\rangle$, and $\langle fx, \mathsf{left}\langle\rangle\rangle$ are well-formed expressions in an appropriate environment, but not values in any environment.

To execute a program *call-by-value* (Plotkin 1975) is to choose computation steps so as to substitute only values for variables, and so as to never look under a variable binding (that is, never examine the body of an abstraction or let-expression) for a subexpression to evaluate. For example, the computation relation that we have been using contains both

$$(\lambda x.\ \mathsf{left}\, x)((\lambda y.\ \mathsf{right}\, y)\langle\rangle) = \mathsf{let}\ (\mathsf{let}\ \langle\rangle\ \mathsf{be}\ y.\ \mathsf{right}\, y)\ \mathsf{be}\ x.\ \mathsf{left}\, x \tag{3.21}$$
$$\triangleright\ (\lambda x.\ \mathsf{left}\, x)(\mathsf{right}\langle\rangle) = \mathsf{let}\ (\mathsf{right}\langle\rangle)\ \mathsf{be}\ x.\ \mathsf{left}\, x$$

and

$$(\lambda x.\ \mathsf{left}\, x)((\lambda y.\ \mathsf{right}\, y)\langle\rangle) = \mathsf{let}\ (\mathsf{let}\ \langle\rangle\ \mathsf{be}\ y.\ \mathsf{right}\, y)\ \mathsf{be}\ x.\ \mathsf{left}\, x \tag{3.22}$$
$$\triangleright\ \mathsf{left}((\lambda y.\ \mathsf{right}\, y)\langle\rangle) = \mathsf{left}(\mathsf{let}\ \langle\rangle\ \mathsf{be}\ y.\ \mathsf{right}\, y)$$

as possible steps from the same program. Call-by-value evaluation eschews (3.22) in favor of (3.21), in order to substitute only values for variables: (3.21) substitutes the value $\langle\rangle$ for the variable y, whereas (3.22) substitutes the nonvalue $(\lambda y.\ \mathsf{right}\, y)\langle\rangle$ for the variable x. To take another example, this computation relation contains both

$$(\lambda y.\ (\lambda x.\ \mathsf{left}\, x)(\mathsf{right}\, y))\langle\rangle = \mathsf{let}\ \langle\rangle\ \mathsf{be}\ y.\ \mathsf{let}\ (\mathsf{right}\, y)\ \mathsf{be}\ x.\ \mathsf{left}\, x \tag{3.23}$$
$$\triangleright\ (\lambda x.\ \mathsf{left}\, x)(\mathsf{right}\langle\rangle) = \mathsf{let}\ (\mathsf{right}\langle\rangle)\ \mathsf{be}\ x.\ \mathsf{left}\, x$$

and

$$(\lambda y.\ (\lambda x.\ \mathsf{left}\, x)(\mathsf{right}\, y))\langle\rangle = \mathsf{let}\ \langle\rangle\ \mathsf{be}\ y.\ \mathsf{let}\ (\mathsf{right}\, y)\ \mathsf{be}\ x.\ \mathsf{left}\, x \tag{3.24}$$
$$\triangleright\ (\lambda y.\ \mathsf{left}(\mathsf{right}\, y))\langle\rangle = \mathsf{let}\ \langle\rangle\ \mathsf{be}\ y.\ \mathsf{left}(\mathsf{right}\, y)$$

as possible steps from the same program. Call-by-value evaluation eschews (3.24) in favor of (3.23), in order to avoid looking under a variable binding for a subexpression to evaluate: (3.23) β-reduces the top-level expression, whereas (3.24) β-reduces a body under a binding for y. In the programming languages in Chapter 2, such choices of computation steps are inconsequential, in the sense that the computation relation is confluent: the programs above all compute to $\mathsf{left}(\mathsf{right}\langle\rangle)$ eventually. But abort destroys confluence, as shown in Section 3.1.1 and below.

$$\#((\lambda x.\ \mathsf{abort}\ 1)(\mathsf{abort}\ 2)) \triangleright \#2 \triangleright 2 \tag{3.25}$$

$$\#((\lambda x.\ \mathsf{abort}\ 1)(\mathsf{abort}\ 2)) \triangleright \#1 \triangleright 1 \tag{3.26}$$

Call-by-value evaluation rules out (3.26) and restores confluence in this case.

To execute a program *left-to-right* is to choose computation steps so as to evaluate subexpressions of a branching expression (that is, an expression of the form FE or $\langle E_1, E_2 \rangle$) from left to right. For example, the computation relation that we have been using contains both

$$\langle (\lambda x.\, x)\langle\rangle, (\lambda x.\, x)\langle\rangle\rangle \rhd \langle\langle\rangle, (\lambda x.\, x)\langle\rangle\rangle \tag{3.27}$$

and

$$\langle (\lambda x.\, x)\langle\rangle, (\lambda x.\, x)\langle\rangle\rangle \rhd \langle (\lambda x.\, x)\langle\rangle, \langle\rangle\rangle \tag{3.28}$$

as possible steps from the same program. Both steps are consistent with call-by-value, but left-to-right evaluation eschews (3.28) in favor of (3.27), in order to evaluate the first component of the ordered pair before the second. To take another example, this computation relation contains both

$$((\lambda f.\, f)(\lambda x.\ \mathsf{left}\, x))((\lambda y.\, \mathsf{right}\, y)\langle\rangle) \rhd (\lambda x.\ \mathsf{left}\, x)((\lambda y.\, \mathsf{right}\, y)\langle\rangle) \tag{3.29}$$

and

$$((\lambda f.\, f)(\lambda x.\ \mathsf{left}\, x))((\lambda y.\, \mathsf{right}\, y)\langle\rangle) \rhd ((\lambda f.\, f)(\lambda x.\ \mathsf{left}\, x))(\mathsf{right}\langle\rangle) \tag{3.30}$$

as possible steps from the same program. Both steps are consistent with call-by-value, but left-to-right evaluation eschews (3.30) in favor of (3.29), in order to evaluate the function before the argument of the top-level application expression. The presence of abort destroys confluence in our language, as shown below.

$$\#((\mathsf{abort}\ 1)(\mathsf{abort}\ 2)) \rhd \#1 \rhd 1 \tag{3.31}$$

$$\#((\mathsf{abort}\ 1)(\mathsf{abort}\ 2)) \rhd \#2 \rhd 2 \tag{3.32}$$

Left-to-right evaluation rules out (3.32) and restores confluence for this program.

Call-by-value, left-to-right evaluation makes computation deterministic (and hence trivially restores confluence): each expression either is in normal form or computes to exactly one expression in a step under such discipline. Figure 3.3 enforces this discipline by restricting the computation relation.

The top part of Figure 3.3 replaces the inference rules for evaluation subcontexts in Figure 3.1 on page 58. The new rules are fewer and stricter: To avoid looking under a variable binding for a subexpression to evaluate, contexts like $\lambda x.\, [\,]$ are no longer allowed. To evaluate subexpressions of a branching expression from left to right, contexts like $F[\,]$ and $\langle E, [\,]\rangle$ are only allowed when F and E are values.

Evaluation subcontexts $D[\,] : d$

$$\frac{}{[\,] : d} \qquad \frac{D[\,] : d \quad \Delta \vdash E : T_1}{D[[\,]E] : d} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V : T_0 \quad D[\,] : d}{D[V[\,]] : d}$$

$$\frac{D[\,] : d \quad \Delta \vdash E : T_2}{D[\langle [\,], E\rangle] : d} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V : T_1 \quad D[\,] : d}{D[\langle V, [\,]\rangle] : d}$$

$$\frac{D[\,] : d}{D[\mathsf{left}\,[\,]] : d} \qquad \frac{D[\,] : d}{D[\mathsf{right}\,[\,]] : d} \qquad \frac{D[\,] : d \quad \Gamma \vdash E' : T'}{D[\mathsf{let}\,[\,]\,\mathsf{be}\,\langle x, y\rangle.\, E'] : d}$$

$$\frac{D[\,] : d \quad \Gamma \vdash E' : T'}{D[\mathsf{let}\,[\,]\,\mathsf{be}\,\langle\rangle.\, E'] : d} \qquad \frac{D[\,] : d \quad \Gamma_1 \vdash E'_1 : T'_1 \quad \Gamma_2 \vdash E'_2 : T'_2}{D[\mathsf{let}\,[\,]\,\mathsf{be}\,\mathsf{left}\,x.\, E'_1 \mid \mathsf{right}\,y.\, E'_2] : d}$$

Computation $E \rhd E'$

$$\begin{aligned}
C[(\lambda x.\, E')V] &\rhd C[E'\,\{x \mapsto V\}] \\
C[\mathsf{let}\,\langle V_1, V_2\rangle\,\mathsf{be}\,\langle x, y\rangle.\, E'] &\rhd C[E'\,\{x \mapsto V_1\}\,\{y \mapsto V_2\}] \\
C[\mathsf{let}\,\langle\rangle\,\mathsf{be}\,\langle\rangle.\, E'] &\rhd C[E'] \\
C[\mathsf{let}\,\mathsf{left}\,V\,\mathsf{be}\,\mathsf{left}\,x.\, E'_1 \mid \mathsf{right}\,y.\, E'_2] &\rhd C[E'_1\,\{x \mapsto V\}] \\
C[\mathsf{let}\,\mathsf{right}\,V\,\mathsf{be}\,\mathsf{left}\,x.\, E'_1 \mid \mathsf{right}\,y.\, E'_2] &\rhd C[E'_2\,\{y \mapsto V\}] \\
C[\#V] &\rhd C[V] \\
C[\#(D[\mathsf{abort}\,E'])] &\rhd C[\#E']
\end{aligned}$$

Figure 3.3. Restricting the computation relation to enforce call-by-value, left-to-right evaluation in a λ-calculus with # and abort

The bottom part of Figure 3.3 replaces the computation relation defined in Section 2.3 on page 30. The new computation relation is smaller: it substitutes only values V, V_1, V_2 for variables x, y, z.

With these restrictions, the recursive function P_2 in (3.10) on page 57 now works reliably on all integer lists. However, at least two fundamental issues remain.

First, we have presented the syntax for # and abort only informally, without providing formal rules that specify when an expression using these constructs is well-formed. It is not trivial to add these inference rules while preserving the crucial property of subject reduction, because we want to classify the same abort expression as well-formed or ill-formed depending on the program it appears in. For example, abort 2 should be allowed in

(3.33) $\quad 1 + \#(\mathsf{let}\,\mathsf{abort}\,2\,\mathsf{be}\,\langle x, y\rangle.\, x + y) \rhd 1 + \#2 \rhd 1 + 2 \rhd 3$

$$\begin{aligned}
\lfloor x \rfloor &= x \\
\lfloor \lambda x.\, E \rfloor &= \lambda x.\, [\![E]\!] \\
\lfloor \langle V_1, V_2 \rangle \rfloor &= \langle \lfloor V_1 \rfloor, \lfloor V_2 \rfloor \rangle \\
\lfloor \langle\rangle \rfloor &= \langle\rangle \\
\lfloor \mathsf{left}\, V \rfloor &= \mathrm{left} \lfloor V \rfloor \\
\lfloor \mathsf{right}\, V \rfloor &= \mathrm{right} \lfloor V \rfloor
\end{aligned}$$

Figure 3.4. The evaluation result $\lfloor V \rfloor$ of values V in the λ-calculus

because the computation concludes in a value, but disallowed in

$$\begin{aligned}
(3.34) \qquad \mathsf{let}\ \#(1 + \mathsf{abort}\ 2)\ \mathsf{be}\ \langle x, y \rangle.\, x + y &\rhd \mathsf{let}\ \#2\ \mathsf{be}\ \langle x, y \rangle.\, x + y \\
&\rhd \mathsf{let}\ 2\ \mathsf{be}\ \langle x, y \rangle.\, x + y
\end{aligned}$$

because the computation gets stuck. What should the type of $\mathsf{abort}\ 2$ be?

Second, to assign a denotational semantics to this programming language that is sound and compositional, we can no longer just let each expression denote its evaluation result. For one thing, any abort expression without a matching # is stuck: it neither is a value nor computes to an expression, so it has no evaluation result to speak of or denote. Yet, replacing one abort expression by another as part of a larger expression can obviously affect the outcome of the larger expression. For example, the outcome of $1 + \#(\mathsf{abort}\ 2)$ is 3, but the outcome of $1 + \#(\mathsf{abort}\ 3)$ is 4. Thus, as in Section 1.4, each expression can no longer just denote its evaluation result.

Strictly speaking, it makes sense to assign denotations to well-formed expressions only once it is defined what well-formed expressions are. For exposition, however, we address denotations first, then return to typing.

3.2. Continuations for delimited control

In this section, we give a well-known denotational semantics and corresponding type system for abort, then generalize them to other control operators. We use the repeatedly rediscovered (Reynolds 1993) concept of *continuations* (Strachey and Wadsworth 1974; Plotkin 1975).

With the addition of # and abort, not every program in our language has an evaluation result; for example, $\mathsf{abort}\langle\rangle$ is stuck. Still, every expression that is a value has an evaluation result; for example, the syntactic expression $\langle\langle\rangle, \langle\rangle\rangle$ results in the semantic pair $\langle\langle\rangle, \langle\rangle\rangle$. We write $\lfloor V \rfloor$ for the evaluation result of a value V, specified in Figure 3.4. The right-hand side of this definition is the λ-calculus of Section 2.3, without # and abort. Although one could program

in this target language, we use it merely as a metalanguage in which to notate evaluation results $\lfloor V \rfloor$ as semantic objects. For example, we write $\langle \mathrm{left}\langle\rangle, \mathrm{right}\langle\rangle\rangle$ in Roman type to mean an ordered pair in the set-theoretic sense. (The second line of Figure 3.4 defines the evaluation result of $\lambda x.\, E$ in terms of the denotation $[\![E]\!]$ of an expression E, which we leave unspecified until Section 3.2.1 below.)

Every program of the form $\#(\ldots)$ also has an evaluation result; for example, $\#(\mathsf{abort}\langle\rangle)$ results in $\langle\rangle$. Every evaluation context $C[\ ]$ thus corresponds to a *continuation* $\lceil C[\ ] \rceil$: a function that maps the evaluation result $\lfloor V \rfloor$ of a value V to the evaluation result of $\#(C[V])$. For example, corresponding to the evaluation context

$$1 + [\ ] + 4 \tag{3.35}$$

is the continuation

$$\lambda x.\, 1 + x + 4, \tag{3.36}$$

the function that maps each number x to $x+5$. (This λ-abstraction is set in Roman type, because the continuation lies in not the syntax but the semantics of the programming language under discussion.) Informally, we may write the type equation

$$\text{continuation} = \text{plugged-value} \to \text{intermediate-answer}. \tag{3.37}$$

For example, the continuation (3.36) is a function from numbers to numbers, reflecting the fact that plugging a numeric value into $\#(1 + [\ ] + 4)$ gives (in other words, sends to the #) a numeric intermediate answer.

For an evaluation subcontext $D[\ ]$, the function $\lceil D[\ ] \rceil$ is sort of a denotation. More precisely, every syntactic rule in the top part of Figure 3.3 on page 63 for building subcontexts $D[\ ]$ corresponds to a semantic rule for building continuations $\lceil D[\ ] \rceil$. For example, the first rule for building subcontexts is that $[\ ]$ is a subcontext.

$$\frac{}{[\ ] : d} \tag{3.38}$$

Because the evaluation result of $\#V$ is V for any value V, the corresponding continuation $\lceil [\ ] \rceil$ is the identity function $\lambda x.\, x$: the plugged value is the same as the evaluation result.

To take a second example, the fifth rule for building subcontexts is that, whenever $D[\ ]$ is a subcontext and V is a value, $D[\langle V, [\ ]\rangle]$ is again a subcontext.

$$\frac{\Gamma \vdash_{\mathrm{V}} V : T_1 \quad D[\ ] : d}{D[\langle V, [\ ]\rangle] : d} \tag{3.39}$$

Given two values V_1 and V_2, the evaluation result of $\#(D[\langle V_1, V_2\rangle])$ is

$$\lceil D[\ ] \rceil \lfloor \langle V_1, V_2 \rangle \rfloor \tag{3.40}$$

by the definition of $\ulcorner D[\ ]\urcorner$. Because

$$\lfloor\langle V_1, V_2\rangle\rfloor = \langle\lfloor V_1\rfloor, \lfloor V_2\rfloor\rangle, \tag{3.41}$$

this evaluation result is also

$$\ulcorner D[\ ]\urcorner\langle\lfloor V_1\rfloor, \lfloor V_2\rfloor\rangle. \tag{3.42}$$

Therefore,

$$\ulcorner D[\langle V_1, [\ ]\rangle]\urcorner\lfloor V_2\rfloor = \ulcorner D[\ ]\urcorner\langle\lfloor V_1\rfloor, \lfloor V_2\rfloor\rangle. \tag{3.43}$$

In other words,

$$\ulcorner D[\langle V_1, [\ ]\rangle]\urcorner = \lambda y.\ulcorner D[\ ]\urcorner\langle\lfloor V_1\rfloor, y\rangle. \tag{3.44}$$

Thus the continuation for $D[\ ]$ and the evaluation result for V_1 together determine the continuation for $D[\langle V_1, [\ ]\rangle]$.

To calculate the continuation for a subcontext that contains a nonvalue expression, such as for a subcontext built using the fourth rule

$$\frac{D[\ ] : d \quad \Delta \vdash E : T_2}{D[\langle [\ ], E\rangle] : d} \tag{3.45}$$

in Figure 3.3 on page 63, we need to assign denotations to nonvalue expressions E. We now turn to these denotations.

3.2.1. The continuation-passing-style transform. As we saw in Section 3.1.1, an abort expression actively takes control over, rather than passively reporting an evaluation result to, its evaluation subcontext. Semantically speaking, it does not provide an argument to the continuation for its evaluation subcontext. To model this denotationally, we follow Strachey and Wadsworth (1974) and let each expression E denote not its evaluation result (which may not even exist) but a function that maps the continuation $\ulcorner D[\ ]\urcorner$ for each evaluation subcontext $D[\ ]$ to the evaluation result of $\#(D[E])$. The idea is that an expression takes a subcontext as argument, whereas a subcontext only takes a value as argument. The rest of this section formalizes this idea.

For example, we let 2 denote not the number 2 but the function $\lambda c.\, c(2)$, because $\ulcorner D[\ ]\urcorner(2)$ is the evaluation result of $\#(D[2])$ for any $D[\ ]$. By contrast, we let abort 2 denote the constant function $\lambda c.\, 2$, because 2 is the evaluation result of $\#(D[\mathsf{abort}\ 2])$ for any $D[\ ]$.

Informally, we may write the type equation (cf. (3.37) on page 65)

$$\text{denotation} = \text{continuation} \to \text{intermediate-answer}. \tag{3.46}$$

Thus, a plugged value and an expression denotation combine semantically in opposite ways with a continuation to give an intermediate answer: whereas a continuation applies to a plugged value to produce an intermediate answer, an expression denotation applies to a continuation to produce an intermediate answer.

$$
\begin{aligned}
[\![V]\!] &= \lambda c.\, c\lfloor V \rfloor \\
[\![\langle E_1, E_2 \rangle]\!] &= \lambda c.\, [\![E_1]\!](\lambda x.\, [\![E_2]\!](\lambda y.\, c\langle x, y\rangle)) \\
[\![\mathsf{left}\, E]\!] &= \lambda c.\, [\![E]\!](\lambda x.\, c(\mathrm{left}\, x)) \\
[\![\mathsf{right}\, E]\!] &= \lambda c.\, [\![E]\!](\lambda x.\, c(\mathrm{right}\, x)) \\
[\![F E]\!] &= \lambda c.\, [\![F]\!](\lambda f.\, [\![E]\!](\lambda x.\, f x c)) \\
[\![\mathsf{let}\ E\ \mathsf{be}\ \langle x, y\rangle.\, E']\!] &= \lambda c.\, [\![E]\!](\lambda v.\, \mathrm{let}\ v\ \mathrm{be}\ \langle x, y\rangle.\, [\![E']\!] c) \\
[\![\mathsf{let}\ E\ \mathsf{be}\ \langle\rangle.\, E']\!] &= \lambda c.\, [\![E]\!](\lambda v.\, \mathrm{let}\ v\ \mathrm{be}\ \langle\rangle.\, [\![E']\!] c) \\
[\![\mathsf{let}\ E\ \mathsf{be}\ \mathsf{left}\, x.\, E'_1 \mid \mathsf{right}\, y.\, E'_2]\!] &= \lambda c.\, [\![E]\!](\lambda v.\, \mathrm{let}\ v\ \mathrm{be\ left}\, x.\, [\![E'_1]\!] c \mid \mathrm{right}\, y.\, [\![E'_2]\!] c) \\
[\![\#E]\!] &= \lambda c.\, c([\![E]\!](\lambda x.\, x)) \\
[\![\mathsf{abort}\, E]\!] &= \lambda c.\, [\![E]\!](\lambda x.\, x)
\end{aligned}
$$

Figure 3.5. The continuation-passing-style transform $[\![E]\!]$ for expressions E in the λ-calculus with # and abort, under call-by-value, left-to-right evaluation

Figure 3.5 makes this idea precise in the form of a translation, called a *continuation-passing-style transform*.[1] As in Figure 3.4, the translation maps the λ-calculus extended with # and abort to the λ-calculus of Section 2.3, without # and abort. As before, although one could program in the target language, we use it merely as a metalanguage in which to notate the meaning $[\![E]\!]$ that our denotational semantics assigns to each source-language expression E. For example, we write $\lambda x.\, x$ in Roman type to mean an identity function in the set-theoretic sense.

Figure 3.5 specifies the denotation $[\![E]\!]$ of an expression E in terms of the evaluation result $\lfloor V \rfloor$ of a value V. (Recall that the translation from V to $\lfloor V \rfloor$ is specified in Figure 3.4; in particular, $\lfloor \lambda x.\, E \rfloor$ is defined to be $\lambda x.\, [\![E]\!]$, so Figures 3.4 and 3.5 are mutually recursive.) Because every value is an expression, $[\![V]\!]$ is defined whenever $\lfloor V \rfloor$ is defined. When V is a value, and $D[\ ]$ is an evaluation subcontext, the evaluation result of $\#(D[V])$ must be equal to $\lceil D[\ ] \rceil \lfloor V \rfloor$ by the definition of $\lceil D[\ ] \rceil$, and to $[\![V]\!] \lceil D[\ ] \rceil$ by the first paragraph of this section. Hence $[\![V]\!] \lceil D[\ ] \rceil$ must be $\lceil D[\ ] \rceil \lfloor V \rfloor$, so we must define $[\![V]\!]$ to be $\lambda c.\, c\lfloor V \rfloor$ for a value V, which we do in the first line of Figure 3.5.

Similar reasoning explains the continuation-passing-style transform for pairs,

$$[\![\langle E_1, E_2 \rangle]\!] = \lambda c.\, [\![E_1]\!](\lambda x.\, [\![E_2]\!](\lambda y.\, c\langle x, y\rangle)). \tag{3.47}$$

The type equations (3.37) and (3.46) say that the denotation of a pair expres-

[1] Strictly speaking, this translation (like its extension in Figure 3.9 on page 76) maps expressions (in particular $\#E$) to continuation-*composing* style (Danvy and Filinski 1990) rather than continuation-passing style.

sion $\langle E_1, E_2\rangle$ should be a function from functions from pairs to answers to answers. To see this, let $D[\ ]$ be a subcontext that is waiting to be plugged with a pair. For any value V_1, the evaluation result of $\#(D[\langle V_1, E_2\rangle])$ must be equal to $[\![E_2]\!]\ulcorner D[\langle V_1, [\ ]\rangle]\urcorner$ by the definition of $[\![E_2]\!]$. Hence

$$\ulcorner D[\langle [\ ], E_2\rangle]\urcorner = \lambda x.\, [\![E_2]\!](\lambda y.\, \ulcorner D[\ ]\urcorner\langle x, y\rangle) \tag{3.48}$$

by the definition of $\ulcorner D[\langle [\ ], E_2\rangle]\urcorner$ and the conclusion (3.44) on page 66. Whether or not E_1 is a value, the evaluation result of $\#(D[\langle E_1, E_2\rangle])$ must be equal to $[\![\langle E_1, E_2\rangle]\!]\ulcorner D[\ ]\urcorner$ by the definition of $[\![\langle E_1, E_2\rangle]\!]$, and to $[\![E_1]\!]\ulcorner D[\langle [\ ], E_2\rangle]\urcorner$ by the definition of $[\![E_1]\!]$. By (3.48), $[\![\langle E_1, E_2\rangle]\!]\ulcorner D[\ ]\urcorner$ must be

$$[\![E_1]\!](\lambda x.\, [\![E_2]\!](\lambda y.\, \ulcorner D[\ ]\urcorner\langle x, y\rangle)), \tag{3.49}$$

so $[\![\langle E_1, E_2\rangle]\!]$ must be

$$\lambda c.\, [\![E_1]\!](\lambda x.\, [\![E_2]\!](\lambda y.\, c\langle x, y\rangle)). \tag{3.50}$$

Figure 3.5 specifies the denotation of some expressions multiple times, consistently. That is, if E is a value of the form $\langle V_1, V_2\rangle$, $\mathsf{left}\, V$, or $\mathsf{right}\, V$, then we can compute $[\![E]\!]$ either using the first line of Figure 3.5 (shown below on the left) or using other lines of Figure 3.5 (shown below on the right).

$$[\![\langle V_1, V_2\rangle]\!] = \lambda c.\, c\lfloor\langle V_1, V_2\rangle\rfloor \qquad [\![\langle V_1, V_2\rangle]\!] = \lambda c.\, [\![V_1]\!](\lambda x.\, [\![V_2]\!](\lambda y.\, c\langle x, y\rangle)) \tag{3.51}$$

$$[\![\mathsf{left}\, V]\!] = \lambda c.\, c\lfloor\mathsf{left}\, V\rfloor \qquad [\![\mathsf{left}\, V]\!] = \lambda c.\, [\![V]\!](\lambda x.\, c(\mathsf{left}\, x)) \tag{3.52}$$

$$[\![\mathsf{right}\, V]\!] = \lambda c.\, c\lfloor\mathsf{right}\, V\rfloor \qquad [\![\mathsf{right}\, V]\!] = \lambda c.\, [\![V]\!](\lambda x.\, c(\mathsf{right}\, x)) \tag{3.53}$$

It is easy to check that the results above on the right β-reduce to those on the left, so the two ways give the same denotation. For example, we can compute the denotation of the source expression

$$\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle \tag{3.54}$$

in two ways (among others). On one hand, because this expression is a value, we can use its evaluation result $\lfloor\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle\rfloor$.

$$\begin{aligned} [\![\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle]\!] &= \lambda c.\, c\lfloor\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle\rfloor \\ &= \lambda c.\, c\langle\lfloor\mathsf{left}\langle\rangle\rfloor, \lfloor\mathsf{right}\langle\rangle\rfloor\rangle \\ &= \lambda c.\, c\langle \mathsf{left}\lfloor\langle\rangle\rfloor, \mathsf{right}\lfloor\langle\rangle\rfloor\rangle \\ &= \lambda c.\, c\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle \end{aligned} \tag{3.55}$$

This denotation is a function from continuations to evaluation results. It makes sense because $\ulcorner D[\ ]\urcorner\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle$ is the result of $\#(D[\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle])$ for any $D[\ ]$. On the other hand, we can also use the continuation-passing-style transform

for pairs.

$$
\begin{aligned}
(3.56)\qquad & [\![\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle]\!] \\
&= \lambda c.\, [\![\mathsf{left}\langle\rangle]\!](\lambda x.\, [\![\mathsf{right}\langle\rangle]\!](\lambda y.\, c\langle x, y\rangle)) \\
&= \lambda c.\, (\lambda c_1.\, c_1 \lfloor \mathsf{left}\langle\rangle \rfloor)(\lambda x.\, (\lambda c_2.\, c_2 \lfloor \mathsf{right}\langle\rangle \rfloor)(\lambda y.\, c\langle x, y\rangle)) \\
&= \lambda c.\, (\lambda c_1.\, c_1(\mathsf{left}\lfloor\langle\rangle\rfloor))(\lambda x.\, (\lambda c_2.\, c_2(\mathsf{right}\lfloor\langle\rangle\rfloor))(\lambda y.\, c\langle x, y\rangle)) \\
&= \lambda c.\, (\lambda c_1.\, c_1(\mathsf{left}\langle\rangle))(\lambda x.\, (\lambda c_2.\, c_2(\mathsf{right}\langle\rangle))(\lambda y.\, c\langle x, y\rangle))
\end{aligned}
$$

We get the same denotation: the last line of (3.56) β-reduces to the last line of (3.55).

The continuation-passing-style transform for pairs is necessary to compute the denotation of a pair whose components are not both values. For example, given

$$
\begin{aligned}
(3.57)\qquad [\![\mathsf{abort}(\mathsf{left}\langle\rangle)]\!] &= \lambda c.\, [\![\mathsf{left}\langle\rangle]\!](\lambda x.\, x) \\
&= \lambda c.\, (\lambda c'.\, c' \lfloor \mathsf{left}\langle\rangle \rfloor)(\lambda x.\, x) \\
&= \lambda c.\ \mathsf{left}\langle\rangle,
\end{aligned}
$$

we can compute the denotation of

$$
(3.58)\qquad \langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle
$$

as follows.

$$
\begin{aligned}
(3.59)\qquad & [\![\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle]\!] \\
&= \lambda c.\, [\![\mathsf{abort}(\mathsf{left}\langle\rangle)]\!](\lambda x.\, [\![\mathsf{abort}(\mathsf{right}\langle\rangle)]\!](\lambda y.\, c\langle x, y\rangle)) \\
&= \lambda c.\, (\lambda c_1.\ \mathsf{left}\langle\rangle)(\lambda x.\, [\![\mathsf{abort}(\mathsf{right}\langle\rangle)]\!](\lambda y.\, c\langle x, y\rangle)) \\
&= \lambda c.\ \mathsf{left}\langle\rangle
\end{aligned}
$$

This denotation is again a function from continuations to evaluation results. It makes sense under left-to-right evaluation, in that $\mathsf{left}\langle\rangle$ is the result of

$$
(3.60)\qquad \#(D[\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle])
$$

for any $D[\ ]$. In particular, when $D[\ ]$ is the null context $[\ ]$, this denotational semantics tells us (correctly) that the outcome of

$$
(3.61)\qquad \#\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle
$$

is $\mathsf{left}\langle\rangle$ rather than $\mathsf{right}\langle\rangle$. The continuation-passing-style transform for pairs thus enforces left-to-right evaluation. For right-to-left evaluation instead, we would change Figure 3.5 to replace (3.47) on page 67 by

$$
(3.62)\qquad [\![\langle E_1, E_2\rangle]\!] = \lambda c.\, [\![E_2]\!](\lambda y.\, [\![E_1]\!](\lambda x.\, c\langle x, y\rangle)).
$$

The expression translation for FE enforces left-to-right evaluation analogously, for example so that

(3.63) $$(\mathsf{abort}(\mathsf{left}\langle\rangle))(\mathsf{abort}(\mathsf{right}\langle\rangle))$$

denotes $\lambda c.\ \mathsf{left}\langle\rangle$ rather than $\lambda c.\ \mathsf{right}\langle\rangle$.

The denotation in (3.59) also enforces call-by-value evaluation: It specifies that the outcome of the program

(3.64) $$\#(\mathsf{let}\ \langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle\ \mathsf{be}\ \langle x, y\rangle.\ \langle\rangle)$$

(which is the expression (3.58) plugged into the evaluation context $\#(\mathsf{let}\ [\ ]\ \mathsf{be}\ \langle x, y\rangle.\ \langle\rangle)$) is $\mathsf{left}\langle\rangle$ rather than $\langle\rangle$. In other words, it prohibits substituting the nonvalues $\mathsf{abort}(\mathsf{left}\langle\rangle)$ and $\mathsf{abort}(\mathsf{right}\langle\rangle)$ for x and y in $\langle\rangle$. If we wanted to relax this prohibition, the denotation of (3.58) would have to be of a different type, such as a pair of functions from continuations rather than a function from continuations to pairs. The other equations in Figure 3.5 similarly embody call-by-value evaluation in the source language (Plotkin 1975; Sabry and Felleisen 1993; Sabry 1994, 1996): for example, the expression

(3.65) $$(\lambda x.\ \mathsf{abort}(\mathsf{left}\langle\rangle))(\mathsf{abort}(\mathsf{right}\langle\rangle))$$

denotes $\lambda c.\ \mathsf{right}\langle\rangle$ rather than $\lambda c.\ \mathsf{left}\langle\rangle$.

Having enriched denotations to be functions from continuations rather than just evaluation results, we can assign meanings to # and abort expressions, as shown in Figure 3.5 and repeated below.

(3.66) $$[\![\#E]\!] = \lambda c.\ c([\![E]\!](\lambda x.\ x))$$

(3.67) $$[\![\mathsf{abort}\ E]\!] = \lambda c.\ [\![E]\!](\lambda x.\ x)$$

These equations are explained by the fact that the null context corresponds to the identity continuation: formally, $\ulcorner[\ ]\urcorner = \lambda x.\ x$. By the definition of $[\![E]\!]$, the evaluation result of $\#E$ is $[\![E]\!](\lambda x.\ x)$. Inspecting the computation relation in Figure 3.3 on page 63 shows that $D[\#E] \rhd D[\#E']$ whenever $\#E \rhd \#E'$, for any expressions E and E' and any subcontext $D[\ ]$. By the definition of $\ulcorner D[\ ]\urcorner$, then, the evaluation result of any $D[\#E]$ is $\ulcorner D[\ ]\urcorner([\![E]\!](\lambda x.\ x))$, so $[\![\#E]\!]$ must be $\lambda c.\ c([\![E]\!](\lambda x.\ x))$. Meanwhile, the equation (3.67) makes sense because $\#(D[\mathsf{abort}\ E])$ computes to $\#E$, whose evaluation result is $[\![E]\!](\lambda x.\ x)$.

To illustrate, we can compute the denotation of

(3.68) $$(\lambda v.\ \mathsf{abort}\langle\rangle)(\#\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle)$$

as follows.

$$
\begin{aligned}
(3.69)\quad [\![\lambda v.\,\mathsf{abort}\langle\rangle]\!] &= \lambda c.\, c\lfloor\lambda v.\,\mathsf{abort}\langle\rangle\rfloor \\
&= \lambda c.\, c(\lambda v.\,[\![\mathsf{abort}\langle\rangle]\!]) \\
&= \lambda c.\, c(\lambda v.\,\lambda c'.\,[\![\langle\rangle]\!](\lambda x.\,x)) \\
&= \lambda c.\, c(\lambda v.\,\lambda c'.\,(\lambda c''.\,c''\lfloor\langle\rangle\rfloor)(\lambda x.\,x)) \\
&= \lambda c.\, c(\lambda v.\,\lambda c'.\,(\lambda c''.\,c''\langle\rangle)(\lambda x.\,x)) \\
&= \lambda c.\, c(\lambda v.\,\lambda c'.\,\langle\rangle)
\end{aligned}
$$

$$
\begin{aligned}
(3.70)\quad [\![\#\langle\mathsf{abort}(\mathsf{left}\langle\rangle),\,\mathsf{abort}(\mathsf{right}\langle\rangle)\rangle]\!] &= \lambda c.\, c([\![\langle\mathsf{abort}(\mathsf{left}\langle\rangle),\,\mathsf{abort}(\mathsf{right}\langle\rangle)\rangle]\!](\lambda x.\,x)) \\
&= \lambda c.\, c((\lambda c'.\ \mathrm{left}\langle\rangle)(\lambda x.\,x)) \\
&= \lambda c.\, c(\mathrm{left}\langle\rangle)
\end{aligned}
$$

$$
\begin{aligned}
(3.71)\quad [\![(\lambda x.\,\mathsf{abort}\langle\rangle)(\#\langle\mathsf{abort}(\mathsf{left}\langle\rangle),\,\mathsf{abort}(\mathsf{right}\langle\rangle)\rangle)]\!] &= \lambda c.\,[\![\lambda x.\,\mathsf{abort}\langle\rangle]\!] \\
&\qquad (\lambda f.\,[\![\#\langle\mathsf{abort}(\mathsf{left}\langle\rangle),\,\mathsf{abort}(\mathsf{right}\langle\rangle)\rangle]\!](\lambda x.\,f\,x\,c)) \\
&= \lambda c.\,(\lambda c_1.\,c_1(\lambda v.\,\lambda c'.\,\langle\rangle))(\lambda f.\,(\lambda c_2.\,c_2(\mathrm{left}\langle\rangle))(\lambda x.\,f\,x\,c)) \\
&= \lambda c.\,\langle\rangle
\end{aligned}
$$

As one expects from call-by-value, left-to-right evaluation, the denotation $\lambda c.\,\langle\rangle$ is equal to that of $\mathsf{abort}\langle\rangle$ alone.

We have seen that adding delimited control to a programming language forces us to revise our denotational semantics, and how to model delimited control using functions from continuations (in other words, functions from functions from results to results to results) as denotations.

3.2.2. Types for continuations. As the target language of our continuation-passing-style transform, the simply-typed λ-calculus provides not just a denotational semantics for the source language but also a type system: via the transform, not only do denotations for the simply-typed λ-calculus pull back to provide denotations for delimited control, but the typing rules for the simply-typed λ-calculus also pull back to provide typing rules for delimited control. In this section, we spell out the latter type system, due to Danvy and Filinski (1989).

Recall from (2.46) on page 41 that types in Section 2.3 are generated by the following context-free grammar.

$$T ::= A \mid T \to T \mid T \times T \mid 1 \mid T + T \tag{3.72}$$

$$A ::= a \mid b \mid c \mid \cdots \tag{3.73}$$

The metavariable T ranges over types; the metavariable A ranges over base types.

Expressions $\Gamma \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)$

$$\frac{\Gamma \vdash_{\mathrm{V}} V : T}{\Gamma \vdash V : W_0 /\!\!/ (T \setminus\!\!\setminus W_0)}\ \text{Value} \qquad \frac{\Gamma \vdash E_1 : W /\!\!/ (T_1 \setminus\!\!\setminus W_1) \quad \Delta \vdash E_2 : W_1 /\!\!/ (T_2 \setminus\!\!\setminus W_0)}{\Gamma, \Delta \vdash \langle E_1, E_2 \rangle : W /\!\!/ ((T_1 \times T_2) \setminus\!\!\setminus W_0)} \times \mathrm{I}$$

$$\frac{\Gamma \vdash E : W /\!\!/ (T_1 \setminus\!\!\setminus W_0)}{\Gamma \vdash \mathsf{left}\, E : W /\!\!/ ((T_1 + T_2) \setminus\!\!\setminus W_0)} + \mathrm{I} \qquad \frac{\Gamma \vdash E : W /\!\!/ (T_2 \setminus\!\!\setminus W_0)}{\Gamma \vdash \mathsf{right}\, E : W /\!\!/ ((T_1 + T_2) \setminus\!\!\setminus W_0)} + \mathrm{I}$$

$$\frac{\Gamma \vdash F : W /\!\!/ ((T_1 \to W_1 /\!\!/ (T_2 \setminus\!\!\setminus W_0)) \setminus\!\!\setminus W_2) \quad \Delta \vdash E : W_2 /\!\!/ (T_1 \setminus\!\!\setminus W_1)}{\Gamma, \Delta \vdash FE : W /\!\!/ (T_2 \setminus\!\!\setminus W_0)} \to \mathrm{E}$$

$$\frac{\Delta \vdash E : W /\!\!/ ((T_1 \times T_2) \setminus\!\!\setminus W_1) \quad \Gamma[x : T_1, y : T_2] \vdash E' : W_1 /\!\!/ (T' \setminus\!\!\setminus W_0)}{\Gamma[\Delta] \vdash \mathsf{let}\ E\ \mathsf{be}\ \langle x, y \rangle.\, E' : W /\!\!/ (T' \setminus\!\!\setminus W_0)} \times \mathrm{E}$$

$$\frac{\Delta \vdash E : W /\!\!/ (1 \setminus\!\!\setminus W_1) \quad \Gamma[\cdot] \vdash E' : W_1 /\!\!/ (T' \setminus\!\!\setminus W_0)}{\Gamma[\Delta] \vdash \mathsf{let}\ E\ \mathsf{be}\ \langle\rangle.\, E' : W /\!\!/ (T' \setminus\!\!\setminus W_0)}\ 1\,\mathrm{E}$$

$$\frac{\begin{array}{c}\Delta \vdash E : W /\!\!/ ((T_1 + T_2) \setminus\!\!\setminus W_1) \\ \Gamma[x : T_1] \vdash E_1' : W_1 /\!\!/ (T' \setminus\!\!\setminus W_0) \\ \Gamma[y : T_2] \vdash E_2' : W_1 /\!\!/ (T' \setminus\!\!\setminus W_0)\end{array}}{\Gamma[\Delta] \vdash \mathsf{let}\ E\ \mathsf{be}\ \mathsf{left}\, x.\, E_1' \mid \mathsf{right}\, y.\, E_2' : W /\!\!/ (T' \setminus\!\!\setminus W_0)} + \mathrm{E}$$

$$\frac{\Gamma \vdash E : W /\!\!/ (T \setminus\!\!\setminus T)}{\Gamma \vdash \#E : W_0 /\!\!/ (W \setminus\!\!\setminus W_0)}\ \# \qquad \frac{\Gamma \vdash E : W /\!\!/ (T \setminus\!\!\setminus T)}{\Gamma \vdash \mathsf{abort}\, E : W /\!\!/ (T_0 \setminus\!\!\setminus W_0)}\ \mathsf{abort}$$

$$\frac{\Gamma[\Delta] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}{\Gamma[\Delta, \Theta] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}\ \text{Weaken} \qquad \frac{\Gamma[\Delta, \Delta] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}{\Gamma[\Delta] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}\ \text{Contract}$$

$$\frac{\Gamma[\Delta, \Theta] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}{\Gamma[\Theta, \Delta] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}\ \text{Exchange} \qquad \frac{\Gamma[(\Delta, \Theta), \Pi] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}{\Gamma[\Delta, (\Theta, \Pi)] \vdash E : W /\!\!/ (T \setminus\!\!\setminus W_0)}\ \text{Associate}$$

Figure 3.6. Defining expressions in the λ-calculus with # and abort

To deal with delimited control operators in the programming language, we change function types from the binary construction $T_1 \to T_2$ to the quaternary construction $T_1 \to W /\!\!/ (T_2 \setminus\!\!\setminus W_0)$, where T_1, W, T_2, W_0 are four types. A type of this form is pronounced "T_1 to W outside T_2 inside W_0". Here T_1 is the usual argument type and T_2 is the usual return type, while W and W_0 are types that record the context in which a function may be applied, as described below. Types T are now generated by the context-free grammar

$$T ::= A \mid T \to T /\!\!/ (T \setminus\!\!\setminus T) \mid T \times T \mid 1 \mid T + T \tag{3.74}$$

Figure 3.6 replaces the judgment form

$$\Gamma \vdash E : T \tag{3.75}$$

for expressions in the simply-typed λ-calculus with a new judgment form

$$\Gamma \vdash E : W /\!\!/ (T \backslash\!\backslash W_0) \tag{3.76}$$

for expressions in our programming language with # and abort. Again, $/\!\!/$ is pronounced "outside", and $\backslash\!\backslash$ is pronounced "inside". This judgment form means that, given any evaluation subcontext $D[\,]$ such that a value of type T can be plugged into $\#(D[\,])$ to yield an evaluation result of type W_0, we can plug the expression E into $\#(D[\,])$ to yield an evaluation result of type W.

The types W_0 and W are known as *answer types*. This terminology reflects the intuition that W_0 and W are types of the evaluation result of an expression of the form $\#(D[E])$. In other words, W_0 and W are types of an intermediate answer provided to a control delimiter. In the simply-typed λ-calculus, the continuation $\lceil D[\,] \rceil$ for the subcontext $D[\,]$ has the type $\lfloor T \rfloor \to \lfloor W_0 \rfloor$, indicating that the evaluation result of $\#(D[E])$ has type W_0 whenever E is a value of type T. For E to satisfy (3.76) means that the evaluation result of $\#(D[E])$ has the type W in the end. Therefore, if E is a value, then the two answer types W_0 and W are the same. Indeed they are in the Value rule in Figure 3.6.

Informally speaking, then, a judgment of the form (3.76) means that E is an expression that behaves locally like the type T, but takes control over its delimited evaluation context, changing the intermediate answer type from W_0 to W. In other words, when we put E inside a subcontext $D[\,]$ that expects an evaluation result of type T so as to produce an intermediate answer of type W_0, the expression E does not report an evaluation result to $D[\,]$ as a value would. Rather, it takes (the continuation for) $D[\,]$ as a semantic argument. Similarly, a function type of the form $T_1 \to W_1 /\!\!/ (T_2 \backslash\!\backslash W_0)$ means that applying the function to something of type T_1 behaves locally like the return type T_2, but takes control over the delimited evaluation context of the application, changing the intermediate answer type from W_0 to W_1.

With this new judgment form for expressions, we also change the $\to \mathrm{I_V}$ rule in Figure 3.2 on page 60 from

$$\frac{\Gamma,\, x : T_1 \vdash E : T_2}{\Gamma \vdash_{\mathrm{V}} \lambda x.\, E : T_1 \to T_2} \to \mathrm{I_V} \tag{3.77}$$

to

$$\frac{\Gamma,\, x : T_1 \vdash E : W /\!\!/ (T_2 \backslash\!\backslash W_0)}{\Gamma \vdash_{\mathrm{V}} \lambda x.\, E : T_1 \to W /\!\!/ (T_2 \backslash\!\backslash W_0)} \to \mathrm{I_V}. \tag{3.78}$$

Here W_0 and W are types that record the context in which the expression E may be evaluated. The expression judgment form (3.76) and the value judgment

form (3.20) on page 60 are designed for the continuation-passing-style transform to translate. To make this statement precise, let us recursively define a map $\lfloor\cdot\rfloor$ from each type T of the form (3.74) to a type $\lfloor T\rfloor$ of the form (3.72).

$$\lfloor A\rfloor = A \tag{3.79}$$

$$\lfloor T_1 \to W /\!\!/ (T_2 \backslash\!\!\backslash W_0)\rfloor = \lfloor T_1\rfloor \to (\lfloor T_2\rfloor \to \lfloor W_0\rfloor) \to \lfloor W\rfloor \tag{3.80}$$

$$\lfloor T_1 \times T_2\rfloor = \lfloor T_1\rfloor \times \lfloor T_2\rfloor \tag{3.81}$$

$$\lfloor 1\rfloor = 1 \tag{3.82}$$

$$\lfloor T_1 + T_2\rfloor = \lfloor T_1\rfloor + \lfloor T_2\rfloor \tag{3.83}$$

Given a type environment Γ where the types are of the form (3.74), we write $\lfloor\Gamma\rfloor$ to mean applying $\lfloor\cdot\rfloor$ to each type in Γ to form a type environment in the simply-typed λ-calculus. The inference rules in Figures 3.2 and 3.6 are reverse-engineered from Figures 3.4 and 3.5 to satisfy two properties that are easy to check by mutual structural induction on source-language derivations.

- Whenever Figure 3.2 (with (3.78) in place of (3.77)) derives $\Gamma \vdash_{\mathrm{V}} V : T$, the simply-typed λ-calculus derives $\lfloor\Gamma\rfloor \vdash \lfloor V\rfloor : \lfloor T\rfloor$.
- Whenever Figure 3.6 derives $\Gamma \vdash E : W /\!\!/ (T \backslash\!\!\backslash W_0)$, the simply-typed λ-calculus derives $\lfloor\Gamma\rfloor \vdash [\![E]\!] : (\lfloor T\rfloor \to \lfloor W_0\rfloor) \to \lfloor W\rfloor$.

For example, let us check the first property for when V is an abstraction: suppose that $V = \lambda x.\, E$ for some x and E. A derivation of $\Gamma \vdash_{\mathrm{V}} V : T$ must conclude with (3.78) or one of the structural rules in Figure 3.2. If the derivation concludes with (3.78), then T is of the form $T_1 \to W /\!\!/ (T_2 \backslash\!\!\backslash W_0)$, and Figure 3.6 must derive

$$\Gamma,\, x : T_1 \vdash E : W /\!\!/ (T_2 \backslash\!\!\backslash W_0). \tag{3.84}$$

By the induction hypothesis, the simply-typed λ-calculus must derive

$$\lfloor\Gamma\rfloor,\, x : \lfloor T_1\rfloor \vdash [\![E]\!] : (\lfloor T_2\rfloor \to \lfloor W_0\rfloor) \to \lfloor W\rfloor. \tag{3.85}$$

The inference

$$\frac{\lfloor\Gamma\rfloor,\, x : \lfloor T_1\rfloor \vdash [\![E]\!] : (\lfloor T_2\rfloor \to \lfloor W_0\rfloor) \to \lfloor W\rfloor}{\lfloor\Gamma\rfloor \vdash \lambda x.\, [\![E]\!] : \lfloor T_1\rfloor \to (\lfloor T_2\rfloor \to \lfloor W_0\rfloor) \to \lfloor W\rfloor} \to \mathrm{I} \tag{3.86}$$

in the simply-typed λ-calculus then derives $\lfloor\Gamma\rfloor \vdash \lfloor V\rfloor : \lfloor T\rfloor$, as desired. If the derivation concludes with a structural rule instead, the corresponding structural rule in the simply-typed λ-calculus completes the induction.

To illustrate this type system, let us derive two sample expressions. First, Figure 3.7a proves that $\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle$ (from (3.54) on page 68) is a well-formed expression. (We omit structural inferences in this proof and those following.) This proof works for any answer type W. The final judgment

$$\cdot \vdash \langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle : W /\!\!/ (((1 + a) \times (b + 1)) \backslash\!\!\backslash W) \tag{3.87}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash_{\mathrm{V}} \langle\rangle : 1}\,1\,\mathrm{I_V}}{\cdot \vdash \langle\rangle : W /\!\!/ (1 \setminus\!\!\setminus W)}\,\mathrm{Value}}{\cdot \vdash \mathsf{left}\langle\rangle : W /\!\!/ ((1+a) \setminus\!\!\setminus W)}\,{+}\mathrm{I} \quad \dfrac{\dfrac{\dfrac{}{\cdot \vdash_{\mathrm{V}} \langle\rangle : 1}\,1\,\mathrm{I_V}}{\cdot \vdash \langle\rangle : W /\!\!/ (1 \setminus\!\!\setminus W)}\,\mathrm{Value}}{\cdot \vdash \mathsf{right}\langle\rangle : W /\!\!/ ((b+1) \setminus\!\!\setminus W)}\,{+}\mathrm{I}}{\cdot \vdash \langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle : W /\!\!/ (((1+a) \times (b+1)) \setminus\!\!\setminus W)}\,{\times}\mathrm{I}$$

(a) Using the Value rule twice at the top

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash_{\mathrm{V}} \langle\rangle : 1}\,1\,\mathrm{I_V}}{\cdot \vdash_{\mathrm{V}} \mathsf{left}\langle\rangle : 1+a}\,{+}\mathrm{I_V} \quad \dfrac{\dfrac{}{\cdot \vdash_{\mathrm{V}} \langle\rangle : 1}\,1\,\mathrm{I_V}}{\cdot \vdash_{\mathrm{V}} \mathsf{right}\langle\rangle : b+1}\,{+}\mathrm{I_V}}{\cdot \vdash_{\mathrm{V}} \langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle : (1+a) \times (b+1)}\,{\times}\mathrm{I_V}}{\cdot \vdash \langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle : W /\!\!/ (((1+a) \times (b+1)) \setminus\!\!\setminus W)}\,\mathrm{Value}$$

(b) Using the Value rule once at the bottom

Figure 3.7. Proving that $\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle$ is well formed

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash_{\mathrm{V}} \langle\rangle : 1}\,1\,\mathrm{I_V}}{\cdot \vdash_{\mathrm{V}} \mathsf{left}\langle\rangle : 1+a}\,{+}\mathrm{I_V}}{\cdot \vdash \mathsf{left}\langle\rangle : (1+a) /\!\!/ ((1+a) \setminus\!\!\setminus (1+a))}\,\mathrm{Value}}{\cdot \vdash \mathsf{abort}(\mathsf{left}\langle\rangle) : (1+a) /\!\!/ (T_1 \setminus\!\!\setminus (b+1))}\,\mathsf{abort} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash_{\mathrm{V}} \langle\rangle : 1}\,1\,\mathrm{I_V}}{\cdot \vdash_{\mathrm{V}} \mathsf{right}\langle\rangle : b+1}\,{+}\mathrm{I_V}}{\cdot \vdash \mathsf{right}\langle\rangle : (b+1) /\!\!/ ((b+1) \setminus\!\!\setminus (b+1))}\,\mathrm{Value}}{\cdot \vdash \mathsf{abort}(\mathsf{right}\langle\rangle) : (b+1) /\!\!/ (T_2 \setminus\!\!\setminus W)}\,\mathsf{abort}}{\cdot \vdash \langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle : (1+a) /\!\!/ ((T_1 \times T_2) \setminus\!\!\setminus W)}\,{\times}\mathrm{I}$$

Figure 3.8. Proving that $\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle$ is well formed

says that, given any subcontext $D[\ ]$ such that a value of type $(1+a) \times (b+1)$ can be plugged into $\#(D[\ ])$ to yield an evaluation result of type W, we can plug the expression $\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle$ into $\#(D[\ ])$ to yield an evaluation result of type W. This conclusion is not surprising because $\langle \mathsf{left}\langle\rangle, \mathsf{right}\langle\rangle\rangle$ is itself a value of type $(1+a) \times (b+1)$. Consequently, we can derive the same judgment using the Value rule just once, as Figure 3.7b does.

Second, Figure 3.8 shows a proof that $\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle$ (from (3.58) on page 69) is well formed. The final judgment says that, given any evaluation context $D[\ ]$ such that a value (of any product type $T_1 \times T_2$) can be plugged into $\#(D[\ ])$ to yield a result (of any type W), we can plug $\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle$ into $\#(D[\ ])$ to yield an evaluation result of type $1+a$. This last type is $1+a$ rather than $b+1$, because the $\times\mathrm{I}$ rule in Figure 3.6 enforces left-to-right

Types T

$$T ::= A \mid T \to T \mathbin{/\!\!/} (T \setminus\!\!\setminus T) \mid T \times T \mid 1 \mid T + T \mid T \setminus\!\!\setminus T$$

Expressions $\Gamma \vdash E : W \mathbin{/\!\!/} (T \setminus\!\!\setminus W_0)$ (additional)

$$\frac{\Gamma, c : T \setminus\!\!\setminus W_0 \vdash E : W \mathbin{/\!\!/} (T \setminus\!\!\setminus W_0)}{\Gamma \vdash \#(cE) : W_0 \mathbin{/\!\!/} (W \setminus\!\!\setminus W_0)}\ \text{Plug} \qquad \frac{\Gamma, c : T_0 \setminus\!\!\setminus W_0 \vdash E : W \mathbin{/\!\!/} (T \setminus\!\!\setminus T)}{\Gamma \vdash \xi c.\, E : W \mathbin{/\!\!/} (T_0 \setminus\!\!\setminus W_0)}\ \text{Shift}$$

Computation $E \rhd E'$ (additional)

$$C[\#(D[\xi c.\, E'])] \rhd C[\#E' \{c[\ ] \mapsto D[\ ]\}]$$

Continuation-passing-style transform $[\![E]\!]$ (additional)

$$[\![\xi c.\, E]\!] = \lambda c.\, [\![E]\!](\lambda x.\, x)$$

Figure 3.9. Abstracting control

evaluation by chaining the answer types W_0, W_1, and W among its conclusion and two premises. In particular, enclosing this expression in #[ ] gives an evaluation result of type $1 + a$ (namely left$\langle\rangle$).

$$\frac{\begin{array}{c}\vdots\ \text{Figure 3.8}\\ \cdot \vdash \langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle : (1 + a) \mathbin{/\!\!/} ((T_1 \times T_2) \setminus\!\!\setminus (T_1 \times T_2))\end{array}}{\cdot \vdash \#\langle \mathsf{abort}(\mathsf{left}\langle\rangle), \mathsf{abort}(\mathsf{right}\langle\rangle)\rangle : W \mathbin{/\!\!/} ((1 + a) \setminus\!\!\setminus W)}\ \# \tag{3.88}$$

3.2.3. Higher-order delimited control. So far, we have

- added # and abort to the syntax of the λ-calculus (in Figure 3.2 on page 60 with (3.78) on page 73 in place of (3.77) on page 73, and in Figure 3.6 on page 72);
- specified their operational semantics using evaluation contexts (in Figure 3.3 on page 63); and
- provided them with a denotational semantics based on continuations (in Figure 3.4 on page 64 and Figure 3.5 on page 67).

This setup supports another delimited control operator, *shift* (Danvy and Filinski 1989, 1990, 1992). Shift is *higher-order* in that it allows passing continuations as functional values (Felleisen 1987, 1988). It subsumes the first-order operator abort, which only discards continuations. We first describe the syntax and semantics of shift, then demonstrate its use in programming examples.

Figure 3.9 adds shift to our programming language. It is a binding construct: the expression $\xi c.\, E$ (pronounced "shift c in E") binds the variable c in the body E. The variable c is special in that it is bound to a subcontext rather than a value. This fact is indicated in the type environment by assigning to c a type of the

(new) form $T \backslash\!\backslash W$, which means that c is a subcontext that can be plugged with a value of type T to produce an intermediate answer of type W. We prohibit the $\mathrm{Id_V}$ rule in Figure 3.2 on page 60 from accessing a variable of such a type in the environment. Instead, the Plug rule must be used, which forces c to appear in the form $\#(cE)$, which means to plug the expression E into the subcontext c. This way, we distinguish between plugging E as a subexpression into a subcontext c and feeding E as an argument to a function c, even though the concrete syntax for Apply and Plug both juxtapose c and E. This new presentation of higher-order delimited control becomes important in Chapter 5, where we introduce a programming language in which an expression may be a subcontext ("coterm") or a subexpression ("term"), whether or not it is a function.

The surrounding # required by the Plug rule is a technical detail to make the small-step operational semantics in Figure 3.9 match the continuation-passing-style denotational semantics (Danvy and Filinski 1989, 1990, 1992). The intuition lies in Section 3.2.1 why we restrict ourselves to plugging an expression E into a subcontext $D[\ ]$ only when $D[\ ]$ is surrounded by #: the continuation for $D[\ ]$ only tells us, given a value V, the evaluation result of $\#(D[V])$, which is surrounded by #. This # distinguishes shift from other delimited control operators, such as Felleisen's *control* (1987, 1988), which is the first delimited control operator in the literature. Continuation-passing-style denotational semantics are also available for these other operators, but they are trickier (Felleisen et al. 1987, 1988; Kiselyov 2005; Shan 2004c; Biernacki et al. 2005a; Dybvig et al. 2005).

The computation step in Figure 3.9 is defined in terms of a new kind of substitution: instead of substituting a value V for a variable x in a body expression E' to yield the new expression $E'\{x \mapsto V\}$, we substitute a context $D[\ ]$ for a variable c in a body expression E' to yield the new expression $E'\{c[\ ] \mapsto D[\ ]\}$. This new kind of substitution is defined just in case c appears in E' only in subexpressions of the form $\#(cE)$, that is, only by the Plug rule. To substitute $D[\ ]$ for c in such E' is to replace each $\#(cE)$ by $\#(D[E])$.

The syntactic rules, operational semantics, and denotational semantics for $\xi c.\, E$ are all identical to those for $\mathsf{abort}\ E$ if we ignore the bound variable c. Indeed, both constructs skip the rest of the computation until the innermost enclosing #, and provide an intermediate answer directly to that #. If the body E never mentions c, then $\xi c.\, E$ degenerates into $\mathsf{abort}\ E$. That is, we can treat $\mathsf{abort}\ E$ as just syntactic sugar for $\xi c.\, E$, where c is a freshly chosen name that does not appear in E.

$$1 + \#(10 \times \xi c.\, 2) \rhd 1 + \#2 \rhd 1 + 2 \rhd 3 \tag{3.89}$$

Unlike $\mathsf{abort}\ E$, the expression $\xi c.\, E$ lets E use the captured evaluation context ($D[\ ]$ in Figure 3.9). For example, $\xi c.\, \#(c2)$ means to capture the current continuation as c, then restore it right away, so $\xi c.\, \#(c2)$ can be replaced by 2

without affecting the outcome of the program. Thus the following computation sequences have the same result.

$$1 + \#(10 \times 2) \rhd 1 + \#20 \rhd 1 + 20 \rhd 21 \tag{3.90}$$

$$\begin{aligned} 1 + \#(10 \times \xi c.\, \#(c2)) &\rhd 1 + \#\#(10 \times 2) \\ &\rhd 1 + \#\#20 \rhd 1 + \#20 \rhd 1 + 20 \rhd 21 \end{aligned} \tag{3.91}$$

The expression $\xi c.\, \#(c2)$ above captures its surrounding delimited context $10 \times [\ ]$ (or semantically, the continuation $\lambda v.\, 10 \times v$) as c when evaluated.

Besides discarding the captured context and reinstating it right away, we can also reuse it later. For example, the following program duplicates a captured context.

$$\begin{aligned} 1 + \#(10 \times \xi c.\, \#(c\#(c2))) &\rhd 1 + \#\#(10 \times \#(10 \times 2)) \\ &\rhd 1 + \#\#(10 \times \#20) \rhd 1 + \#\#(10 \times 20) \\ &\rhd 1 + \#\#200 \rhd 1 + \#200 \rhd 1 + 200 \rhd 201 \end{aligned} \tag{3.92}$$

The captured context can also be reused outside the body E in $\xi c.\, E$. For example, the program below uses $\xi c.\, \lambda v.\, \#(cv)$ to suspend a computation as a function containing a captured context, then resumes it (twice) outside the delimiting #.

$$\begin{aligned} \text{let } \langle 2, \#(10 \times \xi c.\, \lambda v.\, \#(cv)) \rangle \text{ be } \langle x, f \rangle.\, 1 + f(fx) \\ \rhd \text{let } \langle 2, \#(\lambda v.\, \#(10 \times v)) \rangle \text{ be } \langle x, f \rangle.\, 1 + f(fx) \\ \rhd \text{let } \langle 2, \lambda v.\, \#(10 \times v) \rangle \text{ be } \langle x, f \rangle.\, 1 + f(fx) \\ \rhd 1 + (\lambda v.\, \#(10 \times v))((\lambda v.\, \#(10 \times v))2) \\ \rhd 1 + (\lambda v.\, \#(10 \times v))(\#(10 \times 2)) \\ \rhd 1 + (\lambda v.\, \#(10 \times v))(\#20) \\ \rhd 1 + (\lambda v.\, \#(10 \times v))20 \\ \rhd 1 + \#(10 \times 20) \rhd 1 + \#200 \rhd 1 + 200 \rhd 201 \end{aligned} \tag{3.93}$$

We conclude this discussion of delimited control with a more practical programming example for shift: the classical *same-fringe* problem. The goal is to write a (recursive) function F such that, given two binary trees of integers E_1 and E_2, the evaluation result of FE_1E_2 is a Boolean value that indicates whether their fringes are the same.

The input trees E_1 and E_2 are represented by values of type tree int, defined in Section 2.3.5 by the recursive equation (2.53) on page 43:

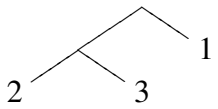
$$\text{tree int} = \text{int} + (\text{tree int} \times \text{tree int}). \tag{3.94}$$

For example, the following two binary trees have the same fringe, namely the sequence $1, 2, 3$.

(3.95) $\mathsf{right}\langle \mathsf{left}\,1, \mathsf{right}\langle \mathsf{left}\,2, \mathsf{left}\,3\rangle\rangle$ $\mathsf{right}\langle \mathsf{right}\langle \mathsf{left}\,1, \mathsf{left}\,2\rangle, \mathsf{left}\,3\rangle$

By contrast, the following binary tree has a different fringe, namely the sequence $2, 3, 1$.

(3.96) $\mathsf{right}\langle \mathsf{right}\langle \mathsf{left}\,2, \mathsf{left}\,3\rangle, \mathsf{left}\,1\rangle$



For clarity, we write leaf and branch in place of left and right when using this encoding of binary trees. For example, the trees in (3.95) are

(3.97) $\mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle\rangle$

and

(3.98) $\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{leaf}\,2\rangle, \mathsf{leaf}\,3\rangle,$

and the tree in (3.96) is

(3.99) $\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle, \mathsf{leaf}\,1\rangle.$

The output Boolean is represented by a value of type $1 + 1$, as mentioned in Section 2.3.4: either $\mathsf{left}\langle\rangle$ if the fringes are different, or $\mathsf{right}\langle\rangle$ if the fringes are the same. For clarity, we write false and true for $\mathsf{left}\langle\rangle$ and $\mathsf{right}\langle\rangle$ when using this encoding of Booleans.

One simple solution to the same-fringe problem is to first compute the fringes of the input trees, then compare the two fringes as lists. To start, we represent a fringe as a value of type $\mathsf{list}\,\mathsf{int}$, defined in Section 2.3.5 by the recursive equation (2.52) on page 42:

(3.100) $$\mathsf{list}\,\mathsf{int} = 1 + (\mathsf{int} \times \mathsf{list}\,\mathsf{int}).$$

For clarity, we write nil and cons for left and right when using this encoding of lists.

The Fringe_1 function below computes the fringe of an input tree. It has the type $\mathsf{tree}\,\mathsf{int} \to W /\!\!/ (\mathsf{list}\,\mathsf{int} \backslash\!\!\backslash W)$ for any answer type W. It is defined in terms of a recursive function Fringe'_1, which takes two arguments.

(3.101) $$\mathit{Fringe}_1 = \lambda t.\ \mathit{Fringe}'_1\ t\ (\mathsf{nil}\langle\rangle)$$

The expression $Fringe'_1\ t\ l$ means to prepend the fringe of the tree t before the list l. Thus $Fringe'_1$ is of type $\mathsf{tree}\,\mathsf{int} \to W /\!\!/ ((\mathsf{list}\,\mathsf{int} \to W' /\!\!/ (\mathsf{list}\,\mathsf{int} \backslash\!\backslash W')) \backslash\!\backslash W)$ for any answer types W and W'.

$$
\begin{aligned}
(3.102)\qquad Fringe'_1 = \lambda t.\, \lambda l.\, &\mathsf{let}\ t\ \mathsf{be}\ \mathsf{leaf}\ x.\ \mathsf{cons}\langle x, l\rangle \\
&\quad \mid \mathsf{branch}\ y.\ \mathsf{let}\ y\ \mathsf{be}\ \langle t_1, t_2\rangle. \\
&\qquad\qquad Fringe'_1\ t_1\ (Fringe'_1\ t_2\ l)
\end{aligned}
$$

The recursive function $Same_1$ below checks whether two lists are equal. It has the type $\mathsf{list}\,\mathsf{int} \to W /\!\!/ ((\mathsf{list}\,\mathsf{int} \to W' /\!\!/ ((1+1) \backslash\!\backslash W')) \backslash\!\backslash W)$ for any answer types W and W'.

$$
\begin{aligned}
(3.103)\quad Same_1 = \lambda l_1.\, \lambda l_2.\, &\mathsf{let}\ l_1\ \mathsf{be}\ \ \mathsf{nil}\ u_1.\ \mathsf{let}\ l_2\ \mathsf{be}\ \mathsf{nil}\ u_2.\ \mathsf{true} \\
&\qquad\qquad\qquad\qquad \mid \mathsf{cons}\ v_2.\ \mathsf{false} \\
&\qquad\qquad \mid\ \mathsf{cons}\ v_1.\ \mathsf{let}\ l_2\ \mathsf{be}\ \mathsf{nil}\ u_2.\ \mathsf{false} \\
&\qquad\qquad\qquad\qquad \mid \mathsf{cons}\ v_2.\ \mathsf{let}\ v_1\ \mathsf{be}\ \langle x_1, y_1\rangle. \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{let}\ v_2\ \mathsf{be}\ \langle x_2, y_2\rangle. \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{if}\ x_1 = x_2 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ Same_1\ y_1\ y_2 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{false}
\end{aligned}
$$

Finally, we can put these two steps together into the desired function F. It has the type $\mathsf{tree}\,\mathsf{int} \to W /\!\!/ ((\mathsf{tree}\,\mathsf{int} \to W' /\!\!/ ((1+1) \backslash\!\backslash W')) \backslash\!\backslash W)$ for any answer types W and W'.

$$
(3.104)\qquad F = \lambda t_1.\, \lambda t_2.\ Same_1 (Fringe_1\ l_1)(Fringe_1\ l_2)
$$

To check this solution against the examples in (3.95) and (3.96), we can compute

$$
\begin{aligned}
(3.105)\qquad & F(\mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle\rangle) \\
& \quad (\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{leaf}\,2\rangle, \mathsf{leaf}\,3\rangle) \\
& \qquad \triangleright^+ Same_1\ (Fringe_1(\mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle\rangle)) \\
& \qquad\qquad\qquad (Fringe_1(\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{leaf}\,2\rangle, \mathsf{leaf}\,3\rangle)) \\
& \qquad \triangleright^+ Same_1\ (\mathsf{cons}\langle 1, \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\langle\rangle\rangle\rangle\rangle) \\
& \qquad\qquad\qquad (Fringe_1(\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{leaf}\,2\rangle, \mathsf{leaf}\,3\rangle)) \\
& \qquad \triangleright^+ Same_1\ (\mathsf{cons}\langle 1, \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\langle\rangle\rangle\rangle\rangle) \\
& \qquad\qquad\qquad (\mathsf{cons}\langle 1, \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\langle\rangle\rangle\rangle\rangle) \\
& \qquad \triangleright^+ \mathsf{true}
\end{aligned}
$$

as well as

$$
\begin{aligned}
(3.106)\qquad & F(\mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle\rangle) \\
& \quad (\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle, \mathsf{leaf}\,1\rangle) \\
& \qquad \triangleright^{+} \mathit{Same}_1\ (\mathit{Fringe}_1(\mathsf{branch}\langle \mathsf{leaf}\,1, \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle\rangle)) \\
& \qquad\qquad\quad (\mathit{Fringe}_1(\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle, \mathsf{leaf}\,1\rangle)) \\
& \qquad \triangleright^{+} \mathit{Same}_1\ (\mathsf{cons}\langle 1, \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\langle\rangle\rangle\rangle\rangle) \\
& \qquad\qquad\quad (\mathit{Fringe}_1(\mathsf{branch}\langle \mathsf{branch}\langle \mathsf{leaf}\,2, \mathsf{leaf}\,3\rangle, \mathsf{leaf}\,1\rangle)) \\
& \qquad \triangleright^{+} \mathit{Same}_1\ (\mathsf{cons}\langle 1, \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\langle\rangle\rangle\rangle\rangle) \\
& \qquad\qquad\quad (\mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{cons}\langle 1, \mathsf{nil}\langle\rangle\rangle\rangle\rangle) \\
& \qquad \triangleright^{+} \mathsf{false}.
\end{aligned}
$$

The test in (3.106) reveals an inefficiency in this solution: even though the two fringes differ at the very beginning (1 is not 2), this solution still computes the rest of the fringes before Same_1 returns false. We want to interleave the computation of the two fringes with their comparison. In other words, we want to represent a fringe as a *lazy list*, or a *stream* (Friedman and Wise 1976). Shift provides one way to program such interleaving, by suspending the two fringe computations and resuming them bit by bit during comparison (Biernacki et al. 2005b,c).[2]

A fringe is either empty or nonempty. The solution above represents a nonempty fringe by an ordered pair whose components are the first element of the fringe (an integer) and the rest of the fringe (a fringe, recursively). We now change this representation to an ordered pair whose components are the first element of the fringe (still an integer) and a suspended computation that, when resumed, produces the rest of the fringe (a function from the unit type to a fringe, recursively). Formally, we represent fringes using the recursive type (cf. (3.100) on page 79)

$$
(3.107)\qquad \text{fringe} = 1 + (\text{int} \times (1 \to W /\!\!/ (\text{fringe} \backslash\!\!\backslash W))),
$$

where W is any answer type. The type of a suspended fringe computation

$$
(3.108)\qquad 1 \to W /\!\!/ (\text{fringe} \backslash\!\!\backslash W)
$$

is the type of $\#(D[\xi c.\, \lambda v.\, \#(c v)])$, if $\#(D[\langle\rangle])$ has the type of a fringe. For clarity, we write snil and scons for left and right when using this suspended encoding of lists.

The Fringe_2 function below computes the fringe of an input tree, using this new representation of fringes. It relies on a recursive function Fringe'_2, which

[2]Generalizing this technique, Kiselyov (2004) lets a suspended computation receive an argument when it is resumed. Using this argument, he implements Huet's *zipper* data-structure (Huet 1997; Hinze and Jeuring 2001; Abbott et al. 2003) in terms of shift.

traverses a given tree depth-first, stopping at each leaf.

(3.109) $$\mathit{Fringe}_2 = \lambda t.\, \#(\mathsf{let}\ \mathit{Fringe}_2'\ t\ \mathsf{be}\ \langle\rangle.\ \mathsf{snil}\langle\rangle)$$

(3.110)
$$\begin{aligned}\mathit{Fringe}_2' = \lambda t.\, \mathsf{let}\ t\ \mathsf{be}\ &\mathsf{leaf}\, x.\, \xi c.\ \mathsf{scons}\langle x, \lambda v.\, \#(c\,v)\rangle\\ &\mid \mathsf{branch}\, y.\, \mathsf{let}\ y\ \mathsf{be}\ \langle t_1, t_2\rangle.\\ &\qquad \mathsf{let}\ \mathit{Fringe}_2'\ t_1\ \mathsf{be}\ \langle\rangle.\\ &\qquad \mathit{Fringe}_2'\ t_2\end{aligned}$$

The Fringe_2' function has the type $\mathsf{tree}\,\mathsf{int} \to \mathsf{fringe} /\!\!/ (1 \backslash\!\!\backslash \mathsf{fringe})$. This means that applying Fringe_2' to a tree behaves locally like the return type 1, but takes control over the delimited evaluation context of the application, whose intermediate answer type is (and stays) fringe. Every time Fringe_2' encounters a $\mathsf{leaf}\, x$, it suspends the computation in c using $\xi c.$, and returns the nonempty fringe $\mathsf{scons}\langle x, \lambda v.\, \#(c\,v)\rangle$ to the innermost enclosing #, namely the # in Fringe_2. The Fringe_2 function has the type $\mathsf{tree}\,\mathsf{int} \to W /\!\!/ (\mathsf{fringe} \backslash\!\!\backslash W)$ for any answer type W. It provides $\mathit{Fringe}_2'\ t$ with its initial delimited evaluation context

(3.111) $$\#(\mathsf{let}\ [\ ]\ \mathsf{be}\ \langle\rangle.\ \mathsf{snil}\langle\rangle),$$

which means that, once Fringe_2' finishes traversing t and returns $\langle\rangle$, the computation of the fringe completes with the empty fringe $\mathsf{snil}\langle\rangle$.

The Same_2 function checks whether two fringes in the new representation are equal. It is the same as Same_1, except crucially the recursive application $\mathit{Same}_1\ y_1\ y_2$ is changed to $\mathit{Same}_2(y_1\langle\rangle)(y_2\langle\rangle)$, so as to resume the suspended computations for the rest of the fringes.

(3.112)
$$\begin{aligned}\mathit{Same}_2 = \lambda l_1.\, \lambda l_2.\, \mathsf{let}\ l_1\ \mathsf{be}\ &\mathsf{snil}\, u_1.\, \mathsf{let}\ l_2\ \mathsf{be}\ \mathsf{snil}\, u_2.\, \mathsf{true}\\ &\qquad \mid \mathsf{scons}\, v_2.\, \mathsf{false}\\ &\quad \mid \mathsf{scons}\, v_1.\, \mathsf{let}\ l_2\ \mathsf{be}\ \mathsf{snil}\, u_2.\, \mathsf{false}\\ &\qquad \mid \mathsf{scons}\, v_2.\, \mathsf{let}\ v_1\ \mathsf{be}\ \langle x_1, y_1\rangle.\\ &\qquad\quad \mathsf{let}\ v_2\ \mathsf{be}\ \langle x_2, y_2\rangle.\\ &\qquad\quad \mathsf{if}\ x_1 = x_2\\ &\qquad\quad \mathsf{then}\ \mathit{Same}_2(y_1\langle\rangle)(y_2\langle\rangle)\\ &\qquad\quad \mathsf{else}\ \mathsf{false}\end{aligned}$$

Putting these pieces together, we can define F in terms of Fringe_2 and Same_2, just as in terms of Fringe_1 and Same_1.

(3.113) $$F = \lambda t_1.\, \lambda t_2.\ \mathit{Same}_2(\mathit{Fringe}_2\ l_1)(\mathit{Fringe}_2\ l_2)$$

Like the old F in (3.104) on page 80, this new F solves the same-fringe problem, but

(3.114) $$F(\mathsf{branch}\langle\mathsf{leaf}\,1, E_1\rangle)(\mathsf{branch}\langle\mathsf{leaf}\,2, E_2\rangle)$$

now computes to false in the same number of computation steps, no matter how large the trees E_1 and E_2 are.

This concludes our presentation of delimited control in programming languages. Before moving to quantification in natural languages, let us stress that shift and reset, especially Danvy and Filinski's type system for them (1989) presented here, are but one instantiation of delimited control. This system has a particularly strong connection to the continuation-passing-style transform, and hence to natural-language quantification as explained below. Nevertheless, other proposals abound (Felleisen 1987, 1988; Felleisen et al. 1987, 1988; Sitaram and Felleisen 1990; Hieb and Dybvig 1990; Queinnec and Serpette 1991; Sitaram 1993; Gunter et al. 1995, 1998) and are also compatible with the transform (Kiselyov 2005; Shan 2004c; Biernacki et al. 2005a; Dybvig et al. 2005). In particular, it can be useful to let an expression name an enclosing control delimiter—not necessarily the innermost one—to capture the current continuation up to.

3.3. Quantification

The following sentences illustrate the linguistic phenomenon of *quantification*.

(3.115) Nobody saw Bob.

(3.116) Some student saw every professor.

(3.117) Alice consulted Bob before most meetings.

As with the previously encountered English sentences, the linguist wants to model which sentences containing words like nobody, some, every, and most are acceptable, and in what situations they are true. In type-logical grammar, it is a non-starter to assign to the *quantificational noun phrase* nobody the type *np* like Alice, for a semantic reason: Unlike Alice, the word nobody does not denote an individual—not a real person; not an imaginary person; not even a concept of a person. In other words, for nobody to denote an individual would make it difficult to model the truth conditions of sentences with nobody.

3.3.1. Generalized quantifiers. We look naturally to first-order predicate logic for inspiration. The beginning logic student is taught to translate a sentence like (3.115) to a universally quantified logical formula like

$$\forall x.\, \neg \mathrm{saw}(x, \mathrm{Bob}), \tag{3.118}$$

which we regard as shorthand for

$$\forall(\lambda x.\, \neg \mathrm{saw}(x, \mathrm{Bob})). \tag{3.119}$$

Here $\lambda x.\, \neg \mathrm{saw}(x, \mathrm{Bob})$ denotes the property of not having seen Bob, a function from individuals to Boolean values. We apply to this property the function $\forall$,

which is a function from properties to truth values. The truth value $\forall(c)$ is true just in case every individual has the property c.

Following the footsteps of the beginning logic student, we want our grammar to derive the expression (3.115) with the meaning (3.118). One way to achieve this goal is to assign to nobody the type $s/(np\backslash s)$, and let it denote the function $\lambda c.\, \forall x.\, \neg cx$. We can then derive (3.115).

$$\dfrac{\mathsf{nobody} \vdash s/(np\backslash s) \qquad \dfrac{\mathsf{saw} \vdash (np\backslash s)/np \quad \mathsf{Bob} \vdash np}{\mathsf{saw},\, \mathsf{Bob} \vdash np\backslash s}\,/\,\mathrm{E}}{\mathsf{nobody},\, (\mathsf{saw},\, \mathsf{Bob}) \vdash s}\,/\,\mathrm{E} \tag{3.120}$$

Unlike the derivation of Alice saw Bob in (2.69) on page 48, this derivation feeds saw Bob to nobody as an argument, rather than applying saw Bob as a function to Alice. Reversing the direction of function application lets us generate the correct denotation (3.118).

This analysis of nobody extends to many other sentences. For example, given that left is a verb phrase like saw Bob, we can derive

(3.121) Nobody left

in the same way.

$$\dfrac{\mathsf{nobody} \vdash s/(np\backslash s) \quad \mathsf{left} \vdash np\backslash s}{\mathsf{nobody},\, \mathsf{left} \vdash s}\,/\,\mathrm{E} \tag{3.122}$$

Other quantificational noun phrases behave similarly to nobody, at least at first glance. For example, we can assign the same type $s/(np\backslash s)$ to the words everyone and someone, but let them denote the functions $\forall$ and $\exists$, respectively. Natural language also expresses quantifiers that cannot be expressed in terms of $\forall$ and $\exists$ in first-order predicate logic, such as "most": We can roughly analyze the phrase most students in

(3.123) Most students saw Bob,

to denote the function that maps each given property c to true just in case the number of students with the property c is more than half of the number of students. A function from properties to truth values (or equivalently, a set of sets of individuals) is called a *generalized quantifier* (Montague 1974b; Barwise and Cooper 1981), or sometimes just "quantifier" for short, because these functions generalize the first-order quantifiers $\forall$ and $\exists$.[3]

To model the difference between nobody and nothing, everyone and everything, and someone and something, we can assume that some individuals are

[3]Often the term "generalized quantifier" is used to refer to a function from tuples of properties (typically pairs of properties) to truth values.

$$\dfrac{\mathsf{Alice} \vdash np \qquad \dfrac{\mathsf{thinks} \vdash (np\backslash s)/s \qquad \dfrac{\mathsf{nobody} \vdash s/(np\backslash s) \qquad \dfrac{\mathsf{saw} \vdash (np\backslash s)/np \quad \mathsf{Bob} \vdash np}{\mathsf{saw}, \mathsf{Bob} \vdash np\backslash s}\,/\mathrm{E}}{\mathsf{nobody}, (\mathsf{saw}, \mathsf{Bob}) \vdash s}\,/\mathrm{E}}{\mathsf{thinks}, (\mathsf{nobody}, (\mathsf{saw}, \mathsf{Bob})) \vdash np\backslash s}\,/\mathrm{E}}{\mathsf{Alice}, (\mathsf{thinks}, (\mathsf{nobody}, (\mathsf{saw}, \mathsf{Bob}))) \vdash s}\,\backslash\mathrm{E}$$

Figure 3.10. Alice thinks nobody saw Bob

animate whereas others are not. To a first approximation, nobody denotes

(3.124) $$\lambda c.\, \forall x.\, \mathrm{animate}(x) \Rightarrow \neg cx$$

(that is, whether every animate individual lacks a given property), whereas nothing denotes

(3.125) $$\lambda c.\, \forall x.\, \neg\mathrm{animate}(x) \Rightarrow \neg cx$$

(that is, whether every inanimate individual lacks a given property). Similarly for everyone versus everything, and someone versus something. The sentence (3.115) on page 83 then denotes

(3.126) $$\forall x.\, \mathrm{animate}(x) \Rightarrow \neg\mathrm{saw}(x, \mathrm{Bob}).$$

When a generalized quantifier is applied to a property, the property (or the expression that denotes it) is called the *scope* of that occurrence of the generalized quantifier. For example, the scope of most students in (3.123) is the property of having seen Bob, or the phrase saw Bob. The scope of nobody in (3.115) is the same. The set of individuals to which a generalized quantifier applies a property, or the expression that denotes the set, is called the *restrictor* of the generalized quantifier. For example, the restrictor of most students is the property of being a student, or the word students. The restrictor of nobody is the property of animacy, or perhaps the suffix -body.

It may appear from the examples so far that the scope of a quantificational noun phrase is always the rest of the sentence containing it, but that is not always the case. For example, the sentence

(3.127) Alice thinks nobody saw Bob

means that Alice thinks the proposition (3.118) on page 83 is true. In this meaning, the scope of the generalized quantifier nobody is the property of having seen Bob (saw Bob), not the property of being thought by Alice to have seen Bob (Alice thinks ... saw Bob). We say that the quantifier nobody *takes scope over* the embedded clause nobody saw Bob, or that it *takes scope at* the boundary between the embedded clause and the rest of the sentence. Figure 3.10 derives this sentence

using the same lexical entry for nobody as proposed above. The / E inferences above apply nobody to its scope saw Bob, and thinks to the resulting meaning of the embedded clause. Hence, as explained in Section 2.4, the formulas-as-types correspondence assigns to this derivation the meaning for (3.127) with the desired scope for nobody.

3.3.2. In-situ quantification. Unfortunately, the simplistic account presented above handles only quantificational noun phrases that combine with their scope to the right. In English and many other languages, quantifiers can occur in a variety of locations throughout a clause. For example, nobody occurs to the right of saw in (3.128), and to the left of 's mother in (3.129) and (3.130).

(3.128) Alice saw nobody.

(3.129) Nobody's mother saw Bob.

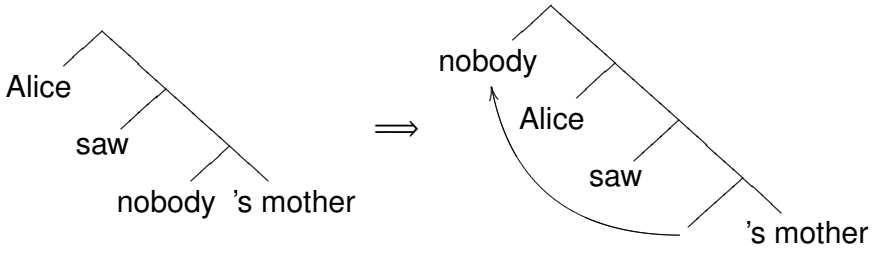
(3.130) Alice saw nobody's mother.

The type $s/(np\backslash s)$ proposed above for nobody can combine neither with saw (of type $(np\backslash s)/np$) on the left nor with 's mother (of type $np\backslash np$) on the right. Therefore, our straw-man analysis predicts erroneously that these sentences are not acceptable.

Roughly speaking, these example sentences suggest that a quantifier in English can occur anywhere in the clause where it takes scope, not just to the left at the top level as in (3.115) on page 83. This flexibility is called *in-situ quantification* because the quantifier can take scope "in place", surrounded by its scope.

Starting with Montague's seminal Proper Treatment of Quantification (1974b), many ways to account for in-situ quantification have been proposed in the natural-language syntax and semantics literature. Some proposals, such as May's Logical Form (1985) and Hobbs and Shieber's (1987) and Moran's (1988) quantifier scoping algorithms, posit a level of representation where quantifiers *move* silently to the edge of their scopes. For example, we could move nobody in (3.130) as sketched below.

(3.131)



The meaning of the sentence can then be computed from the latter representation, where the quantifier nobody does occur to the left at the top level. Other proposals, such as Cooper storage (1983) and Keller storage (1988), enrich the mechanism for semantic interpretation to deal specifically with quantification. For example,

$$\dfrac{\mathsf{nobody} \vdash q(np,s,s) \qquad \dfrac{\mathsf{Alice} \vdash np \qquad \dfrac{\mathsf{saw} \vdash (np\backslash s)/np \qquad \dfrac{\dfrac{}{np \vdash np}\,\mathrm{Id} \qquad \mathsf{'s\ mother} \vdash np\backslash np}{np,\ \mathsf{'s\ mother} \vdash np}\,/\mathrm{E}}{\mathsf{saw},\ (np,\ \mathsf{'s\ mother}) \vdash np\backslash s}\,\backslash\mathrm{E}}{\mathsf{Alice},\ (\mathsf{saw},\ (np,\ \mathsf{'s\ mother})) \vdash s}\,\backslash\mathrm{E}}{\mathsf{Alice},\ (\mathsf{saw},\ (\mathsf{nobody},\ \mathsf{'s\ mother})) \vdash s}\,q\,\mathrm{E}$$

Figure 3.11. Alice saw nobody's mother

Hendriks's Flexible Types system (1988) systematically generates an infinite family of types to account for different surroundings in which a quantifier may wake up from the lexicon to find itself.

Particularly relevant to us is Moortgat's attractive proposal in type-logical grammar of a ternary type constructor q (1988, 1995, 1996). If T, W_0, and W are types, then Moortgat proposes that $q(T, W_0, W)$ be a type as well. Moortgat terms T the *bound expression* type, W_0 the *binding domain* type, and W the *resultant expression* type. A quantificational noun phrase canonically has the type $q(np, s, s)$. In terms of semantics, an expression of type $q(T, W_0, W)$ denotes a function from functions from T to W_0 to W, or formally, $(T \to W_0) \to W$. In terms of syntax, an expression of type $q(T, W_0, W)$ can plug into any context into which an expression of type T can be plugged to produce a larger expression of type W_0, to produce a larger expression of type W. That is, we add the inference rule

$$\dfrac{\Delta \vdash F : q(T, W_0, W) \quad \Gamma[x : T] \vdash E : W_0}{\Gamma[\Delta] \vdash F(\lambda x.\, E) : W}\,q\,\mathrm{E}. \tag{3.132}$$

With the type $q(np, s, s)$ for nobody and the same denotation as above, we can derive the previously problematic sentences. For instance, Figure 3.11 derives (3.130). (We give examples below where W_0 and W differ.)

The $q\,\mathrm{E}$ rule in (3.132) suggests the following intuition behind the three types T, W_0, and W in the type $q(T, W_0, W)$. An expression of type $q(T, W_0, W)$ behaves "locally" like type T. That is, it combines syntactically with phrases (in $\Gamma[\ ]$) with which a T can combine. But "behind the scenes", it waits until syntactic combination yields a W_0. It then takes scope over that W_0 to produce a W finally. We call W_0 and W the *answer types* in the type $q(T, W_0, W)$.

The W produced when a quantificational subexpression of type $q(T, W_0, W)$ takes scope enters the rest of the derivation like any other W. In particular, another quantificational subexpression (of type $q(T', W, W')$, say) may take scope over it. Hence we predict that multiple quantifiers may occur in the same sentence and take scope over the same clause. Many examples bear out this prediction. For example, we empirically observe that the sentence

(3.133) A man is robbed in New York every 11 seconds

is ambiguous between two *readings* due to the quantificational noun phrases a man and every 11 seconds: one where a possibly different man is robbed each time, and one where the same man is repeatedly robbed. The follow-up sentence

(3.134) Let's interview him

disambiguates (3.133) in favor of the second reading. For simplicity, we consider the sentence

(3.135) Someone saw everyone,

which is similarly ambiguous between two readings due to the quantificational noun phrases someone and everyone. The *linear scope* (or *surface scope*) reading corresponds to the logical formula

$$\exists x.\, \forall y.\, \mathrm{saw}(x, y), \tag{3.136}$$

which we regard as shorthand for

$$\exists(\lambda x.\, \forall(\lambda y.\, \mathrm{saw}(x, y))). \tag{3.137}$$

To describe this reading, we say that someone takes *wide scope* over everyone, which takes *narrow scope*. The *inverse scope* reading corresponds to the logical formula

$$\forall y.\, \exists x.\, \mathrm{saw}(x, y), \tag{3.138}$$

which we regard as shorthand for

$$\forall(\lambda y.\, \exists(\lambda x.\, \mathrm{saw}(x, y))). \tag{3.139}$$

We say that everyone takes inverse scope over someone in this reading because the order between the quantifiers in the English sentence is the opposite of that in the corresponding logical formula.

Our grammar correctly predicts that (3.135) is acceptable and ambiguous: it provides two derivations for (3.135) that assign different denotations. Figure 3.12a derives the linear-scope reading. As the two successive qE inferences indicate, someone applies to the clausal meaning produced by everyone. Figure 3.12b derives the inverse-scope reading, where everyone applies to the clausal meaning produced by someone instead.

So far, we have only used q at the type $q(np, s, s)$. The same formal mechanism for in-situ quantification, be it q or another account, turns out to be applicable to a wide range of linguistic phenomena, where the answer type is not always s. In other words, if we generalize generalized quantifiers beyond those that take scope over s to yield s, then many other linguistic phenomena reduce to in-situ quantification. For example, Morrill (2000, 2003) proposes an analysis of anaphora that essentially treats a pronoun and its antecedent both as quantifiers of

$$\dfrac{\textsf{someone} \quad \dfrac{\textsf{everyone} \quad \dfrac{}{np \vdash np}\,\mathrm{Id} \quad \dfrac{\textsf{saw} \vdash (np\backslash s)/np \quad \dfrac{}{np \vdash np}\,\mathrm{Id}}{\textsf{saw}, np \vdash np\backslash s}\,/\mathrm{E}}{np, (\textsf{saw}, np) \vdash s}\,\backslash\mathrm{E}}{\textsf{someone}, (\textsf{saw}, \textsf{everyone}) \vdash s}\,q\mathrm{E}$$

(a) Linear scope

$$\dfrac{\textsf{everyone} \quad \dfrac{\textsf{someone} \quad \dfrac{}{np \vdash np}\,\mathrm{Id} \quad \dfrac{\textsf{saw} \vdash (np\backslash s)/np \quad \dfrac{}{np \vdash np}\,\mathrm{Id}}{\textsf{saw}, np \vdash np\backslash s}\,/\mathrm{E}}{np, (\textsf{saw}, np) \vdash s}\,\backslash\mathrm{E}}{\textsf{someone}, (\textsf{saw}, \textsf{everyone}) \vdash s}\,q\mathrm{E}$$

(b) Inverse scope

Figure 3.12. Someone saw everyone

a sort. We return to this idea in Section 4.5.1, though without Morrill's assumption (problematic as explained in Section 2.4) that syntactic combination is associative.

To take another example, Carpenter (1994) and Morrill (1994) encode intensionality with q. Given our discussion of intensionality in Section 1.5, we can understand their idea as follows. Suppose that the type np denotes the set of individuals, including planets. Then the verb thinks shows that the noun phrases the morning star and the evening star cannot be of type np (and denote the same planet). But if the type s denotes not the (two-element) set of truth values but the set of functions from possible worlds to truth values (perhaps s is shorthand for $w \to s_0$, where w denotes the set of possible worlds and s_0 denotes the set of truth values), then we can assign to the morning star and the evening star the type $q(np, s, s)$. The denotation of the morning star would be

(3.140) $\lambda c.\, \lambda w.\, c(\text{the morning star in the world } w)(w)$,

whereas the denotation of the evening star would be

(3.141) $\lambda c.\, \lambda w.\, c(\text{the evening star in the world } w)(w)$.

An intensional verb, such as thinks, can then distinguish between these two denotations. Informally speaking, the morning star and the evening star take scope over a clause to access its possible world.

Although q is versatile, its status as a logical connective raises concerns. We have only provided the qE rule, which specifies how to consume a quantificational expression (Δ in (3.132) on page 87). It is unclear what a corresponding qI rule might be, with which to create a quantificational expression. Given the intuition of plugging a subexpression into a context, we expect the judgments

$$T \vdash q(T, W, W) \tag{3.142}$$

and

$$T'/T,\ q(T, W, W) \vdash q(T', W, W) \tag{3.143}$$

to be logical validities. Yet neither is derivable using qE alone, simply because q appears to the right of $\vdash$ in these judgments.

Concomitant with this proof-theoretic concern about q is a model-theoretic one. Several different sets of inference rules for q have been proposed (Moortgat 1996; Barker 2005). What makes any given set of inference rules for q correct? (That is, with respect to what class of semantic models do we want a set of inference rules for q that is sound and complete?)

One reason why the empirical linguist cares for judgments like (3.142) and (3.143) to be valid arises in the analysis of and. It would be nice to be able to say that and can conjoin two phrases just in case they have the same type. This natural idea explains why

(3.144) Alice saw Bob's mother and Carol

is acceptable (because Bob's mother and Carol both have type *np*), but not

(3.145) *Alice saw Bob's mother and to Carol's assassination

(because to Carol's assassination does not have type *np*). Given that

(3.146) Alice saw Bob and someone's mother

is acceptable, we would like Bob and someone's mother to have the same type. Using (3.142) and (3.143), we can derive the type $q(np, s, s)$ for Bob and someone's mother respectively, as desired.

One way to strengthen the foundation for q is to reduce it to other logical connectives that are better understood. Bernardi (2003) summarizes three implementations of q in multimodal type-logical grammar (Morrill 1994; Moortgat 1995, 2000); and Areces and Bernardi (2003) give a fourth solution using hybrid logic. In Chapter 4, we propose yet another implementation of q, which unlike the others is motivated by a computational (or operational, or dynamic, or processing) interpretation based on continuations.

3.4. Delimited control versus quantification

Delimited control in programming languages and in-situ quantification in natural languages exemplify the same general pattern as in Section 1.6: the program abort 2 does not evaluate to any result; the utterance nobody does not refer to any individual. Yet we would like a sound and compositional theory that explains how these expressions can take part in a complete program or utterance that does evaluate to a result or refer to an individual or a truth value.

Programming-language theorists and linguists alike have responded to this challenge by adding to denotations a new aspect of meaning beyond evaluation results and reference, namely to let an expression take control over, or apply to, its context. The meaning of the context is known as the *continuation* of an expression in programming-language semantics and the *scope* of a quantifier in natural-language semantics. With this addition, denotations become more complex: they are now functions from functions from a value type T to an answer type W_0 to an answer type W.[4] The scope of a natural-language quantifier is analogous to the delimited context of a programming-language control operator (Barker 2001, 2002; de Groote 2001). The analogy is as if nobody denotes the shift-expression $\xi c.\, \forall x.\, \neg\#(cx)$ (Shan 2005, 2004a; Barker 2004).

Ambiguity may result when a program contains multiple control operators or an utterance contains multiple quantifiers. In Section 3.2, the denotation of the control operator evaluated earlier applies to a continuation containing the denotation of the control operator evaluated later. In Section 3.3.2, the denotation of the quantifier taking wide scope applies to a scope containing the denotation of the quantifier taking narrow scope. Thus evaluation order in programming languages corresponds to scope order in natural languages: left-to-right evaluation (as in the denotation (3.47) on page 67) corresponds to linear scope (as in the denotation (3.137) on page 88); right-to-left evaluation (as in the denotation (3.62) on page 69) corresponds to inverse scope (as in the denotation (3.139) on page 88) (Barker 2001, 2002; de Groote 2001). Section 3.1.2 above removes nonconfluence in a programming language by enforcing left-to-right evaluation. Sections 4.4 and 4.5 below explain empirical generalizations in linguistics, also by enforcing left-to-right evaluation.

A popular view in programming-language theory holds that each computational side effect corresponds to a *notion of computation* expressed as a *monad* or *monad morphism* (Moggi 1990, 1991; Wadler 1992a,b). Under this view, delimited control turns out to subsume all computational side effects (Filinski 1994, 1996, 1999). Given our analogy between computational and linguistic side

[4] The traditional answer type is s in linguistics but $\bot$ in logic and computer science, where undelimited control was studied before delimited control. De Groote (2001) relates in-situ quantification to undelimited control by identifying the types s and $\bot$.

effects, we then expect quantification to subsume all linguistic side effects (Shan 2001). Section 4.5 below confirms this expectation by using quantification to account for a variety of linguistic side effects observed empirically.

CHAPTER 4

Evaluation order in natural languages

This chapter implements in-situ quantification in multimodal type-logical grammar using an idea from the analysis of delimited control in programming languages: the meaning of an evaluation context is a continuation (Section 3.2). In type-logical grammar, Moortgat's ternary type constructor q for in-situ quantification (1988; 1996; 1995) already enjoys several proposed implementations (Morrill 1994; Moortgat 1995, 2000), summarized by Bernardi (2003). Moreover, both Barker (2001, 2002, 2004) and de Groote (2001) already relate quantification to continuations. Nevertheless, our implementation improves over previous linguistic theories in two ways, one theoretical and one empirical.

First, whereas previous implementations of q in type-logical grammar are designed to move a quantifier to the edge of its scope as depicted in (3.131) on page 86, our implementation does not move nearby constituents apart or distant constituents together. Rather, as Section 4.1 explains, the structural rules that we propose just encode symmetries inherent in a syntactic structure viewed as a graph. In Section 4.2, we use this encoding to model how a quantifier such as nobody combines with its scope outward, and how a scope such as Alice saw [ ]'s mother combines with its gap inward, just as an intransitive verb such as slept combines with its subject argument leftward, and as a transitive verb such as saw combines with its object argument rightward. (Chapter 5 exploits the same symmetries for programming-language theory.)

Second, we treat the order in which a human processes parts of an utterance like the order in which a computer evaluates parts of a program. Just as Section 3.1.2 specifies the evaluation order of a toy programming language by restricting its evaluation contexts, Sections 4.3 and 4.4 specify the processing order of a natural-language fragment by restricting its scope-taking contexts. In Section 4.5, we use this new notion of order in natural language to unify the preference for linear scope in quantification with empirical generalizations in other linguistic side effects: crossover in anaphora, superiority in wh-questions, and the effect of linear order on polarity sensitivity. In particular, Section 4.5.2 shows how this unified account of crossover and wh-questions predicts a complex pattern of interaction between them, without any stipulation that mentions both anaphora and interrogation. Section 4.5.3 then describes the first concrete processing account of linear order in polarity sensitivity, linking it to linear scope in

quantification.

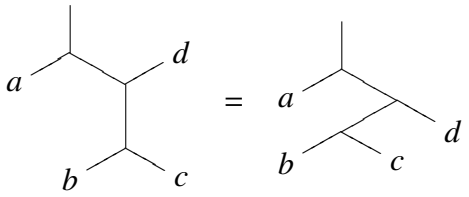
4.1. Type environments as graphs

This section introduces a graphical interpretation of type environments in type-logical grammar, in terms of graphs. Section 4.2 then applies this interpretation to implement *q* for linguistic analyses.

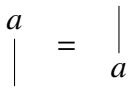
Every type environment in type-logical grammar can be uniquely drawn as a binary tree. It is crucial for us to view the root of the tree as just another leaf node. Because this is not a conventional view of syntactic structure, let us be pedantically specific and define a binary tree to be an acyclic graph in which every node has either one or three edges, and every leaf node but one (the *root*) is labeled with a type. For example, the type environment

$$\Gamma = a, ((b, c), d) \tag{4.1}$$

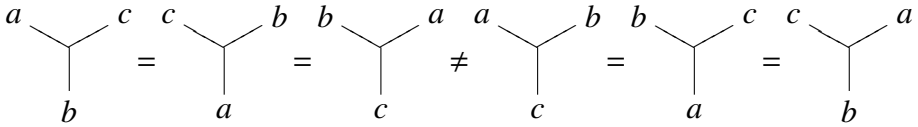
can be drawn as follows.

(4.2) $a, ((b, c), d) \iff$ 

In our graphs, edges are unoriented, so the following two graphs are identical.

(4.3) 

However, we do orient nodes by designating, for each trivalent node, one of the two cyclic orderings of its edges as counterclockwise. Hence each of the following graphs are equal to two others, but not to their mirror images.

(4.4) 

Together, these two conventions ensure that each type environment corresponds to only one graph.

Suppose now that Γ and Θ are two type environments, such that Θ appears as part of Γ. For instance, Γ might be the type environment $a, ((b, c), d)$ as above, and Θ might be the type environment b, c. We write $\Gamma = \Delta[\Theta]$, where $\Delta[\ ]$ is informally the context in Γ of the subexpression Θ; that is, $\Delta[\ ] = a, ([\ ], d)$. Graphically speaking, we have decomposed Γ into two parts, $\Delta[\ ]$ and Θ, by

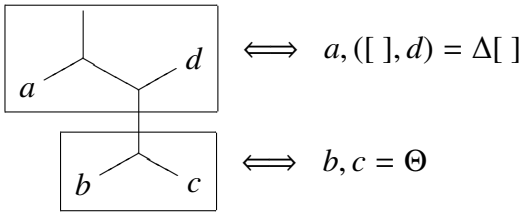
$$\frac{\Gamma \circledcirc x : T_1 \vdash E : T_2}{\Gamma \vdash \lambda x.\, E : T_2 /\!\!/ T_1}\ /\!\!/\,\mathrm{I} \qquad \frac{x : T_1 \circledcirc \Gamma \vdash E : T_2}{\Gamma \vdash \lambda x.\, E : T_1 \backslash\!\!\backslash T_2}\ \backslash\!\!\backslash\,\mathrm{I}$$

$$\frac{\Gamma \vdash F : T_2 /\!\!/ T_1 \quad \Delta \vdash E : T_1}{\Gamma \circledcirc \Delta \vdash FE : T_2}\ /\!\!/\,\mathrm{E} \qquad \frac{\Delta \vdash E : T_1 \quad \Gamma \vdash F : T_1 \backslash\!\!\backslash T_2}{\Delta \circledcirc \Gamma \vdash FE : T_2}\ \backslash\!\!\backslash\,\mathrm{E}$$

Figure 4.1. Adding the binary mode ⊚ to the Lambek calculus without products

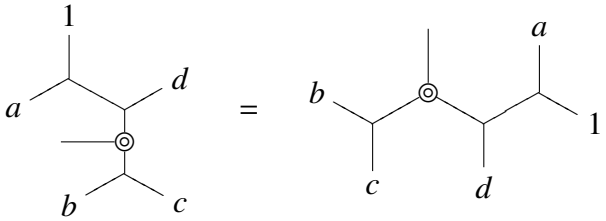
ripping apart the graph at an edge, calling the side that contains the root node $\Delta[\ ]$, and the other side Θ.

(4.5)



To represent such a decomposition as a graph, we insert a new, special node ⊚ in the middle of the edge that connects the two components.

(4.6)



This new node ⊚ connects the two components of the decomposition to a new root node. The old root node becomes a leaf with the special label 1. We know which side of the ⊚ corresponds to the context, since that is the side that contains the 1 node marking the position of the old root. Starting at the ⊚ node, we always put the context side right after the subexpression side (where "after" means moving counterclockwise). The symbol ⊚ is pronounced "at", because it puts a subexpression *at* the hole of a context.

To convert the picture (4.6) back to a type environment, we introduce a new binary mode (following Section 2.4.1), the *continuation* or *context* mode. Instead of writing this new mode's comma and slashes with subscripts as $,_c$, $/_c$, and $\backslash_c$, we use the easier-to-read symbols ⊚ (again, pronounced "at"), $/\!\!/$ (pronounced "outside"), and $\backslash\!\!\backslash$ (pronounced "inside"). The new inference rules, following Figure 2.28 on page 51, are shown in Figure 4.1. The graph in (4.6) thus corresponds to the formula

(4.7) $$(b, c) \circledcirc (d, (1, a)).$$

In the terminology of Section 2.4.1, this mode is internal (or unpronounceable).

The continuation mode $\circledcirc$ treats the decomposition $\Gamma = \Delta[\Theta]$ as a type environment in its own right: if Δ is the type environment that represents the context $\Delta[\ ]$ (for example, $\Delta = d, (1, a)$ above), then $\Theta \circledcirc \Delta$ decomposes Γ into the subexpression Θ and the context $\Delta[\ ]$. In the trivial case, the graph is ripped apart at the root edge into two sides: the null context $[\ ]$, and the type environment Γ viewed as its own (improper) subexpression. Because the null context is represented by the type environment 1, we henceforth treat 1 as the *right identity* for $\circledcirc$. That is, we introduce the structural rule[1]

$$\frac{\Gamma[\Theta] \vdash T}{\Gamma[\Theta \circledcirc 1] \vdash T}\ \text{Root} \tag{4.8}$$

to make $\Theta \circledcirc 1$ logically equivalent to just Θ. Informally, this rule mandates the equation

$$\Theta = \Theta \circledcirc 1 \tag{4.9}$$

in an algebra of subexpressions and contexts.

In the original type environment $\Gamma = a, ((b, c), d)$ in (4.1) on page 94, the type d follows a and combines with b, c, whereas in the decomposition $(b, c) \circledcirc (d, (1, a))$ in (4.7), the type d precedes a and combines with $1, a$. It appears from (4.7) that Δ represents a context $\Delta[\ ]$ by turning it *inside-out*, but the graphical view in (4.6) shows that we have simply inserted a continuation node $\circledcirc$ into an edge, without pulling nearby constituents far apart or pushing distant constituents close together. We have merely changed our perspective by placing one subexpression in the foreground and backgrounding the rest—in Belnap's words (1982), *displaying* a subexpression. We can pronounce the environment $d, (1, a)$ as "a hole d", because it is just what the context $a, ([\ ], d)$ looks like from the perspective of the continuation node. In general, if an environment built up using the default mode (the comma ,) contains a unique 1, then we can pronounce it as what is to the right of the 1, followed by "hole", followed by what is to the left of the 1. Interpreting environments as graphs makes mapping between contexts and environments as simple as rotating a graph as in (4.6).

Every implementation of in-situ quantification in multimodal type-logical grammar represents a meta-level notion of context (like $\Delta[\ ] = a, ([\ ], d)$) as an object-level expression (like $\Delta = d, (1, a)$), in order to let an expression access (that is, take scope over) its context. We want to represent contexts at the object level, in order to preserve our logical machinery with its desirable metatheoretic properties (such as the *finite-reading property*: any expression can only be ambiguous among a finite number of meanings). Our implementation is unique

[1] Restall (2000; pages 30–31) calls this rule Push when it is used to infer from top to bottom, and Pop when it is used to infer from bottom to top.

in representing contexts "inside-out", just as the grammars for evaluation contexts in Chapter 2 (starting with Figure 2.9 on page 27) represents contexts "inside-out". This view results from *defunctionalizing* (Reynolds 1972) continuations (Danvy and Nielsen 2001a,b; Danvy 2004), and corresponds to Huet's *zipper* data-structure (Huet 1997; Hinze and Jeuring 2001; Abbott et al. 2003) for the data type of binary trees. Section 4.5 uses this connection between contexts in natural and programming languages to analyze empirical observations on natural language.

We now introduce two structural rules to equate equivalent decompositions of the same graph. If we find some type environment of the form Θ, Π inside a context $\Delta[\ ]$ (that is, given the type environment $\Delta[\Theta, \Pi]$), we can move either Θ or Π from the subexpression part to the context part of the decomposition. In other words, the environment $\Delta[\Theta, \Pi]$ can be decomposed in three ways: one that isolates Θ, one that isolates Θ, Π, and one that isolates Π. If the type environment Δ represents the context $\Delta[\ ]$ (so that it contains 1 in place of the root of $\Delta[\Theta, \Pi]$), then Figure 4.2 depicts these three decompositions. Since these are three decompositions of the same graph, we want them to be logically equivalent, so we add two structural rules.[2]

$$\text{(4.10)} \qquad \frac{\Gamma[\Theta \circledcirc (\Pi, \Delta)] \vdash T}{\Gamma[(\Theta, \Pi) \circledcirc \Delta] \vdash T}\ \text{Left} \qquad\qquad \frac{\Gamma[(\Theta, \Pi) \circledcirc \Delta] \vdash T}{\Gamma[\Pi \circledcirc (\Delta, \Theta)] \vdash T}\ \text{Right}$$

Informally speaking, these structural rules mandate the equation

$$\text{(4.11)} \qquad \Theta \circledcirc (\Pi, \Delta) = (\Theta, \Pi) \circledcirc \Delta = \Pi \circledcirc (\Delta, \Theta)$$

[2]Unfortunately, in the presence of the two structural rules in (4.10) and the right identity 1 for the $\circledcirc$ mode, the default mode becomes commutative.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma[\Theta, \Pi] \vdash T}{\Gamma[(\Theta, \Pi) \circledcirc 1] \vdash T}\ \text{Root}}{\Gamma[\Theta \circledcirc (\Pi, 1)] \vdash T}\ \text{Left}}{\Gamma[\Theta \circledcirc ((\Pi, 1) \circledcirc 1)] \vdash T}\ \text{Root}}{\Gamma[\Theta \circledcirc (\Pi \circledcirc (1, 1))] \vdash T}\ \text{Left}}{\Gamma[\Theta \circledcirc ((1, \Pi) \circledcirc 1)] \vdash T}\ \text{Right}}{\Gamma[\Theta \circledcirc (1, \Pi)] \vdash T}\ \text{Root}}{\Gamma[(\Pi, \Theta) \circledcirc 1] \vdash T}\ \text{Right}}{\Gamma[\Pi, \Theta] \vdash T}\ \text{Root}$$

As explained in Section 2.4, we do not want the default mode to commute, because word order is significant in natural language. In Section 4.4 below, to enforce left-to-right evaluation, we restrict the Right rule to apply only when the environment Θ is surrounded by $\diamond$. With this restriction, the Right inferences above (especially the upper one) no longer apply. (Barker and Shan (2005) mention two other ways to prevent the default mode from commuting.) Therefore, we leave the commutativity issue aside here.

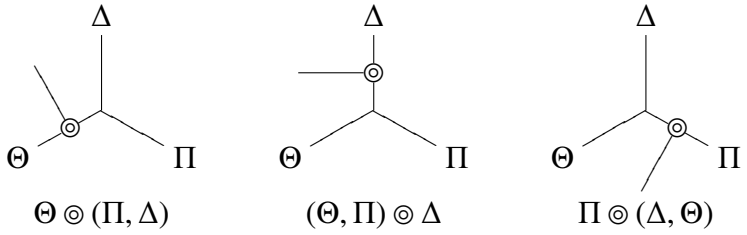


Figure 4.2. Three ways to decompose $\Delta[\Theta, \Pi]$ into a subexpression and a context

to relate the two modes , and ⊙ in an algebra of subexpressions and contexts. Using these structural rules, any decomposition of a (connected) graph Θ can be derived from the trivial decomposition $\Theta \odot 1$, which represents Θ itself inside the null context [ ] and is logically equivalent to just Θ by (4.8) on page 96. For example, the decomposition in (4.6) on page 95 can be derived in three steps.[3]

$$\dfrac{\dfrac{\dfrac{\Gamma[a, ((b, c), d)] \vdash T}{\Gamma[(a, ((b, c), d)) \odot 1] \vdash T}\,\text{Root}}{\Gamma[((b, c), d) \odot (1, a)] \vdash T}\,\text{Right}}{\Gamma[(b, c) \odot (d, (1, a))] \vdash T}\,\text{Left} \tag{4.12}$$

The continuation mode ⊙ and its associated structural rules expand the machinery of abstraction and application built-in to type-logical grammar beyond describing functions that apply leftward or rightward. We can now describe functions that apply "inward" or "outward". Such functions underlie in-situ quantification, as Section 4.2 below shows.

4.2. Scopes apply inward; quantifiers apply outward

Let us apply the techniques developed above to types and expressions from natural language. As in Sections 2.1 and 2.4, we assume a type of noun phrases (more precisely, proper nouns) *np* and a type of sentences (more precisely, clauses) *s*. For example, Alice saw Bob is a clause: Syntactically, the phrase saw combines with Bob to the right, then Alice to the left, using the default mode (the comma ,). Semantically, the function saw takes Bob and Alice as arguments.

$$\dfrac{\mathsf{Alice} \vdash np \qquad \dfrac{\mathsf{saw} \vdash (np\backslash s)/np \quad \mathsf{Bob} \vdash np}{\mathsf{saw}, \mathsf{Bob} \vdash np\backslash s}\,/\,\text{E}}{\mathsf{Alice}, (\mathsf{saw}, \mathsf{Bob}) \vdash s}\,\backslash\,\text{E} \tag{4.13}$$

[3] Whereas the steps for splay-tree rotation (Sleator and Tarjan 1985) move among different trees that parenthesize the same fringe, our steps in (4.10) move among different decompositions of the same tree.

$$\dfrac{\dfrac{a : \mathsf{Alice} \vdash a : np \qquad \dfrac{f : \mathsf{saw} \vdash f : (np\backslash s)/np \qquad \dfrac{}{x : np \vdash x : np}\,\mathrm{Id}}{f : \mathsf{saw}, x : np \vdash fx : np\backslash s}\,/\,\mathrm{E}}{a : \mathsf{Alice}, (f : \mathsf{saw}, x : np) \vdash fxa : s}\,\backslash\,\mathrm{E}}{\dfrac{(a : \mathsf{Alice}, (f : \mathsf{saw}, x : np)) \circledcirc 1 \vdash fxa : s}{(f : \mathsf{saw}, x : np) \circledcirc (1, a : \mathsf{Alice}) \vdash fxa : s}\,\mathrm{Right}}{\dfrac{x : np \circledcirc ((1, a : \mathsf{Alice}), f : \mathsf{saw}) \vdash fxa : s}{(1, a : \mathsf{Alice}), f : \mathsf{saw} \vdash \lambda x.\, fxa : np \backslash\!\!\backslash s}\,\backslash\!\!\backslash\,\mathrm{I}}\,\mathrm{Root}\,\mathrm{Right}$$

Figure 4.3. Alice saw [ ]

In general, any two noun phrases with saw in between form a clause.

$$\dfrac{\dfrac{}{np \vdash np}\,\mathrm{Id} \qquad \dfrac{\mathsf{saw} \vdash (np\backslash s)/np \qquad \dfrac{}{np \vdash np}\,\mathrm{Id}}{\mathsf{saw}, np \vdash np\backslash s}\,/\,\mathrm{E}}{np, (\mathsf{saw}, np) \vdash s}\,\backslash\,\mathrm{E} \tag{4.14}$$

The context Alice saw [ ] has a hole [ ] where an *np* can be plugged in to yield an *s*. In linguistic terms, it is a gapped clause. Following the graphical approach in Section 4.1, we represent this context as a type environment, or equivalently, a graph.

(4.15) $(1, \mathsf{Alice}), \mathsf{saw} \iff$

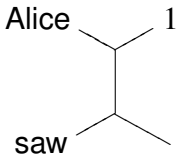


Figure 4.3 uses the structural rules in (4.10) on page 97 to prove that this context is a gapped clause, in other words, that the type environment in (4.15) entails the type $np \backslash\!\!\backslash s$. The latter type is the type of something that gives *s* when combined on the left with an *np* using $\circledcirc$. In other words, a gapped clause is a context that encloses an *np* to give an *s*.

Figure 4.3 shows the proof with meanings (λ-expressions to the left of :). As explained in Section 2.4, the formulas-as-types correspondence determines these meanings from the syntactic derivation: the / E and \ E inferences correspond to function application; while the $\backslash\!\!\backslash$ I inference at the bottom corresponds to abstraction. Thus the meaning of the gapped clause Alice saw [ ] is the function $\lambda x.\, fxa$.

We can now treat quantificational noun phrases like everyone. As a syntactic element, everyone combines with a gapped clause enclosing it to give a complete

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots\ \text{Figure 4.3 on page 99}}{e : \mathsf{everyone} \vdash e : s/\!\!/(np\backslash\!\!\backslash s)} \qquad \dfrac{}{(1, a : \mathsf{Alice}), f : \mathsf{saw} \vdash \lambda x.\, f x a : np\backslash\!\!\backslash s}}{e : \mathsf{everyone} \circledcirc ((1, a : \mathsf{Alice}), f : \mathsf{saw}) \vdash e(\lambda x.\, f x a) : s}\ /\!\!/\,\mathrm{E}}{(f : \mathsf{saw}, e : \mathsf{everyone}) \circledcirc (1, a : \mathsf{Alice}) \vdash e(\lambda x.\, f x a) : s}\ \text{Right}}{(a : \mathsf{Alice}, (f : \mathsf{saw}, e : \mathsf{everyone})) \circledcirc 1 \vdash e(\lambda x.\, f x a) : s}\ \text{Right}}{a : \mathsf{Alice}, (f : \mathsf{saw}, e : \mathsf{everyone}) \vdash e(\lambda x.\, f x a) : s}\ \text{Root}$$

Figure 4.4. Alice saw everyone

clause. In terms of our context mode $\circledcirc$, everyone gives s when combined on the right with $np\backslash\!\!\backslash s$. Thus we assign to it the type $s/\!\!/(np\backslash\!\!\backslash s)$. Figure 4.4 uses this type to prove the grammaticality of Alice saw everyone. This proof assigns the denotation $e(\lambda x.\, f x a)$ to the sentence. This denotation is the desired outcome under the standard view that the meaning e of everyone is a generalized quantifier that maps the property of being seen by Alice to the proposition that Alice saw everyone.

Just as the function-type constructors / and \ for the default mode , take arguments from the right and from the left, the function-type constructors $/\!\!/$ and $\backslash\!\!\backslash$ for the continuation mode $\circledcirc$ take arguments from outside (that is, apply to surrounding contexts) and from inside (that is, apply to enclosed subexpressions), respectively. More generally, we can reconstruct the ternary type constructor q for in-situ quantification in type-logical grammar: encode $q(A, B, C)$ as $C/\!\!/(A\backslash\!\!\backslash B)$. The type for everyone in Figure 4.4 encodes $q(np, s, s)$, as expected.

Informally speaking, Figure 4.3 derives the "judgment"

(4.16) $$\mathsf{Alice\ saw}\ [\ ] \vdash np\backslash\!\!\backslash s$$

using a bureaucracy of structural rules. Omitting this bureaucracy from Figure 4.4 yields the "derivation" of Alice saw everyone in Figure 4.5.

As noted in Section 3.3.2, the English sentence

(4.17) Someone saw everyone

is ambiguous between a linear-scope reading

(4.18) $$\exists x.\, \forall y.\, \mathrm{saw}(x, y)$$

and an inverse-scope reading

(4.19) $$\forall y.\, \exists x.\, \mathrm{saw}(x, y).$$

Like any account of in-situ quantification implementing q, our system derives both readings for the sentence, as shown in Figures 4.6 and 4.7. Ignoring the structural rules Root, Left, and Right, these derivations correspond exactly to

$$\dfrac{\mathsf{everyone} \vdash s /\!\!/ (np \backslash\!\!\backslash s) \qquad \dfrac{\vdots}{\mathsf{Alice\ saw}\ np \vdash s}}{\mathsf{Alice\ saw\ everyone} \vdash s}\ \begin{array}{l}\backslash\!\!\backslash \mathrm{I} \\ /\!\!/ \mathrm{E}\end{array}$$

Figure 4.5. Alice saw everyone, without the formal bureaucracy

$$\dfrac{\exists{:}\mathsf{someone} \vdash \exists{:} s /\!\!/ (np \backslash\!\!\backslash s) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots\ (4.14)\ \text{on page 99}}{x{:}np, (f{:}\mathsf{saw}, y{:}np) \vdash fyx{:}s}\ \mathrm{Root}}{(x{:}np, (f{:}\mathsf{saw}, y{:}np)) \circledcirc 1 \vdash fyx{:}s}\ \mathrm{Right}}{(f{:}\mathsf{saw}, y{:}np) \circledcirc (1, x{:}np) \vdash fyx{:}s}\ \mathrm{Right}}{y{:}np \circledcirc ((1, x{:}np), f{:}\mathsf{saw}) \vdash fyx{:}s}\ \backslash\!\!\backslash \mathrm{I}}{(1, x{:}np), f{:}\mathsf{saw} \vdash \lambda y.\, fyx{:}np \backslash\!\!\backslash s}\ /\!\!/ \mathrm{E}}{\forall{:}\mathsf{everyone} \circledcirc ((1, x{:}np), f{:}\mathsf{saw}) \vdash \forall(\lambda y.\, fyx){:}s}\ \mathrm{Right}}{(f{:}\mathsf{saw}, \forall{:}\mathsf{everyone}) \circledcirc (1, x{:}np) \vdash \forall(\lambda y.\, fyx){:}s}\ \mathrm{Right}}{(x{:}np, (f{:}\mathsf{saw}, \forall{:}\mathsf{everyone})) \circledcirc 1 \vdash \forall(\lambda y.\, fyx){:}s}\ \mathrm{Left}}{x{:}np \circledcirc ((f{:}\mathsf{saw}, \forall{:}\mathsf{everyone}), 1) \vdash \forall(\lambda y.\, fyx){:}s}\ \backslash\!\!\backslash \mathrm{I}}{(f{:}\mathsf{saw}, \forall{:}\mathsf{everyone}), 1 \vdash \lambda x. \forall(\lambda y.\, fyx){:}np \backslash\!\!\backslash s}\ /\!\!/ \mathrm{E}}{\exists : \mathsf{someone} \circledcirc ((f : \mathsf{saw}, \forall : \mathsf{everyone}), 1) \vdash \exists(\lambda x.\, \forall(\lambda y.\, fyx)) : s}\ \mathrm{Left}}{(\exists : \mathsf{someone}, (f : \mathsf{saw}, \forall : \mathsf{everyone})) \circledcirc 1 \vdash \exists(\lambda x.\, \forall(\lambda y.\, fyx)) : s}\ \mathrm{Root}}{\exists : \mathsf{someone}, (f : \mathsf{saw}, \forall : \mathsf{everyone}) \vdash \exists(\lambda x.\, \forall(\lambda y.\, fyx)) : s}$$

Figure 4.6. Linear scope for Someone saw everyone

ones using q in a system where q is built-in, such as with the $q\,\mathrm{E}$ rule in (3.132). The quantifier that takes wider scope is the one that takes its context as argument lower in the proof.

The derivations in this section illustrate the rule of 1 in our system. In formal logic, 1 is a nullary mode, just as the empty type environment $\cdot$ in Section 2.3 is a nullary mode, and as the default mode , and the continuation mode $\circledcirc$ are two binary modes. The computational analogue of $\Theta \circledcirc 1$ is to enclose a programming-language expression (corresponding to Θ) in a control delimiter: On one hand, the top-to-bottom direction of the Root rule says that $\Theta \circledcirc 1$ entails Θ. Using Root to infer from top to bottom, we can prove that the null context 1 denotes the identity

$$\dfrac{\forall\!:\!\mathsf{everyone}\vdash \forall\!:\!s/\!\!/(np\backslash\!\!\backslash s)\quad \dfrac{(1,\exists\!:\!\mathsf{someone}),f\!:\!\mathsf{saw}\vdash \lambda y.\exists(\lambda x.fyx)\!:\!np\backslash\!\!\backslash s}{y\!:\!np\circledcirc((1,\exists\!:\!\mathsf{someone}),f\!:\!\mathsf{saw})\vdash \exists(\lambda x.fyx)\!:\!s}\backslash\!\!\backslash\mathrm{I}}{\dfrac{(f\!:\!\mathsf{saw},y\!:\!np)\circledcirc(1,\exists\!:\!\mathsf{someone})\vdash \exists(\lambda x.fyx)\!:\!s}{\dfrac{(\exists\!:\!\mathsf{someone},(f\!:\!\mathsf{saw},y\!:\!np))\circledcirc 1\vdash \exists(\lambda x.fyx)\!:\!s}{\dfrac{\exists\!:\!\mathsf{someone}\circledcirc((f\!:\!\mathsf{saw},y\!:\!np),1)\vdash \exists(\lambda x.fyx)\!:\!s}{\dfrac{\exists\!:\!\mathsf{someone}\vdash \exists\!:\!s/\!\!/(np\backslash\!\!\backslash s)\quad (f\!:\!\mathsf{saw},y\!:\!np),1\vdash \lambda x.fyx\!:\!np\backslash\!\!\backslash s}{x\!:\!np\circledcirc((f\!:\!\mathsf{saw},y\!:\!np),1)\vdash fyx\!:\!s}/\!\!/\mathrm{E}}\backslash\!\!\backslash\mathrm{I}}\,\mathrm{Left}}\,\mathrm{Left}}\,\mathrm{Right}}\,\mathrm{Right}}\,\mathrm{Root}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall : \mathsf{everyone} \circledcirc ((1, \exists : \mathsf{someone}), f : \mathsf{saw}) \vdash \forall(\lambda y.\ \exists(\lambda x.\ fyx)) : s}{(f : \mathsf{saw}, \forall : \mathsf{everyone}) \circledcirc (1, \exists : \mathsf{someone}) \vdash \forall(\lambda y.\ \exists(\lambda x.\ fyx)) : s}\,\mathrm{Right}}{(\exists : \mathsf{someone}, (f : \mathsf{saw}, \forall : \mathsf{everyone})) \circledcirc 1 \vdash \forall(\lambda y.\ \exists(\lambda x.\ fyx)) : s}\,\mathrm{Right}}{\exists : \mathsf{someone}, (f : \mathsf{saw}, \forall : \mathsf{everyone}) \vdash \forall(\lambda y.\ \exists(\lambda x.\ fyx)) : s}\,\mathrm{Root}$$

Figure 4.7. Inverse scope for Someone saw everyone

continuation $\lambda x.\ x$, of type $T\backslash\!\!\backslash T$ for any T, as follows.

$$\dfrac{\dfrac{\dfrac{}{x : T \vdash x : T}\,\mathrm{Id}}{x : T \circledcirc 1 \vdash x : T}\,\mathrm{Root}}{1 \vdash \lambda x.\ x : T\backslash\!\!\backslash T}\backslash\!\!\backslash\mathrm{I} \tag{4.20}$$

On the converse hand, the bottom-to-top direction of the Root rule says that Θ entails $\Theta \circledcirc 1$, so Θ can apply outward to the null context. In other words, Root from bottom to top inserts an expression in a control delimiter (like the # rule in Figure 3.6 on page 72), while Root from top to bottom removes a control delimiter (like the computation steps for # in (3.12) on page 57 and Figure 3.3 on page 63). Together, the two directions of Root in this natural-language fragment correspond to the null-context inference rule

$$\dfrac{}{[\ ]} \tag{4.21}$$

in the small-step operational semantics in Chapters 2 and 3.

Whereas Root manages the null context 1, the other structural rules Left and Right manage contexts of other forms. More precisely, Left manages contexts of the form $\Delta[[\ ], \Pi]$, and Right manages contexts of the form $\Delta[\Theta, [\ ]]$, where Θ and Π are two (environments that represent) expressions. If we were to define

contexts in the style of (4.21), Left and Right would correspond respectively to

$$\frac{\Delta[\ ]}{\Delta[[\ ], \Pi]} \qquad\qquad \frac{\Delta[\ ]}{\Delta[\Theta, [\ ]]}. \tag{4.22}$$

In words, a context may descend recursively into either branch of a constituent built with the default mode.

The rules (4.21) (Root) and (4.22) (Left and Right) specify a notion of context that lets [ ] appear anywhere in a type environment built up using the default mode. In other words, they allow a graph to be decomposed at any edge. Hence this treatment of gapped clauses and quantification (unlike several other type-logical treatments) works uniformly no matter where the gap or quantifier appears in the clause over which it takes scope: at the left as in Everyone saw Bob, at the right as in Alice saw everyone, or in the middle as in Alice introduced everyone to Bob. It also does not matter how or how deeply the gap or quantifier is embedded. In particular, our system handles quantificational noun phrases in possessive position without further stipulation, as in Everyone's mother saw Bob. Every type-logical treatment of quantification we are aware of (that is more sophisticated than the straw-man proposal in Section 3.3.1) enjoys this success, but it is by no means typical of linguistic frameworks other than type-logical grammar, which associate syntax with semantics less tightly.

Compared to other type-logical treatments of quantification (especially other implementations of *q*), ours uniquely represents the scope of a quantifier "inside-out", as explained in Section 4.1. We deal with in-situ quantification not by moving the quantifier to the edge of its scope, as is standard, but by viewing the same graph from the perspective of the continuation node. This view makes it easy to come up with the structural rules Root, Left, and Right above as well as the refinements and generalizations in the sections below.

4.3. Refining the analysis of quantification by refining notions of context

This section extends the analysis of quantification above to account for two kinds of additional empirical observations. First, we discuss situations in which one quantificational noun phrase contains another, as in every dean of a school. The challenge is to understand how every can take scope over (or in computational terms, be evaluated before) something it contains. Second, we discuss *scope islands*: expressions that limit the scope of quantifiers they contain. For instance, it is generally assumed that everyone in Someone thought everyone left cannot take scope outside of the embedded clause everyone left, and hence cannot take inverse scope over someone. If so, then the embedded clause is a scope island, and the challenge is to enforce its islandhood.

In both cases, the empirical observations we discuss are well studied in the literature. Our contribution is to relate them to the operational semantics of

programming languages: We restrict what a quantificational expression can take scope over by restricting what we allow as a legitimate context that the expression can access. This is the same strategy by which Section 3.1.2 enforces evaluation order in a toy programming language, and by which Section 4.4 below enforces evaluation order in a natural-language fragment.

As explained in Section 4.2 above, the structural rules Root, Left, and Right specify a notion of context by relating the continuation mode ⊚ to other modes. We can change these rules to change what is allowed as a context: adding structural rules broadens the notion to include more scope-taking possibilities; removing structural rules constrains the notion to include fewer scope-taking possibilities. (Different notions of context can be mixed in the same grammar by using one ⊚-like mode for each notion, though we have no need to do so here.) We demonstrate each linguistic application of this flexibility by making a separate amendment to the same basic system presented above. These amendments are compatible in the sense that it is straightforward to combine them into one grammar that covers all of the applications at once.

4.3.1. Restrictors. Having treated simple quantificational noun phrases like everyone and someone in Section 4.2, we now turn to quantifiers that combine with a common noun (like school) or a more complex restrictor constituent, as in most schools or every dean of a school. (For simplicity, we ignore the difference between plural and singular nouns, like schools and school. We also treat dean of as a single word.) We assume that common nouns are predicates, in that they denote functions from individuals (type *np*) to propositions or truth values (type *s*). However, they cannot directly combine with noun phrases in the default mode—that is why *Harvard school is unacceptable; it does not mean that Harvard is a school. Thus, as promised in Section 2.4.1, we introduce a new binary mode *n* (the letter *n* being mnemonic for "noun") and assign common nouns to the type $np\backslash_n s$.[4] (For intuition, we can imagine that is a in Harvard is a

[4]We cannot reuse the continuation mode ⊚ for this new mode and assign common nouns to the type $np \backslash\!\backslash s$. If we did, then the Right_n rule that we are about to add in (4.26) would predict incorrectly that the strings Alice is a dean of Harvard and *Harvard is an Alice dean of are equally acceptable and have the same meaning, as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Alice} \circledcirc (\text{dean of}, \text{Harvard})}{(\text{Alice} \circledcirc (\text{dean of}, \text{Harvard})) \circledcirc 1}\,\text{Root}}{((\text{Alice}, \text{dean of}) \circledcirc \text{Harvard}) \circledcirc 1}\,\text{Left}}{\text{Harvard} \circledcirc (1 \circledcirc (\text{Alice}, \text{dean of}))}\,\text{Right}_n}{\text{Harvard} \circledcirc ((1, \text{Alice}) \circledcirc \text{dean of})}\,\text{Left}}{\text{Harvard} \circledcirc (\text{Alice} \circledcirc (\text{dean of}, 1))}\,\text{Right}}{\text{Harvard} \circledcirc ((\text{Alice}, \text{dean of}) \circledcirc 1)}\,\text{Left}}{\text{Harvard} \circledcirc (\text{Alice}, \text{dean of})}\,\text{Root}$$

$$\dfrac{\dfrac{\dfrac{e : \mathsf{every} \vdash e : (s/\!\!/(np\backslash\!\!\backslash s))/(np\backslash_n s) \qquad \dfrac{\dfrac{\dfrac{}{x : np \vdash x : np}\,\mathrm{Id} \qquad g : \mathsf{griped} \vdash g : np\backslash s}{x : np, g : \mathsf{griped} \vdash gx : s}\,\backslash\mathrm{E}}{(x : np, g : \mathsf{griped}) \circledcirc 1 \vdash gx : s}\,\mathrm{Root}}{x : np \circledcirc (g : \mathsf{griped}, 1) \vdash gx : s}\,\mathrm{Left}}{g : \mathsf{griped}, 1 \vdash \lambda x.\, gx : np\backslash\!\!\backslash s}\,\backslash\!\!\backslash\mathrm{I}}{s : \mathsf{school} \vdash s : np\backslash_n s}\,/\mathrm{E}}{e : \mathsf{every}, s : \mathsf{school} \vdash es : s/\!\!/(np\backslash\!\!\backslash s)}\,/\!\!/\mathrm{E}$$

$$\dfrac{\dfrac{(e : \mathsf{every}, s : \mathsf{school}) \circledcirc (g : \mathsf{griped}, 1) \vdash es(\lambda x.\, gx) : s}{((e : \mathsf{every}, s : \mathsf{school}), g : \mathsf{griped}) \circledcirc 1 \vdash es(\lambda x.\, gx) : s}\,\mathrm{Left}}{(e : \mathsf{every}, s : \mathsf{school}), g : \mathsf{griped} \vdash es(\lambda x.\, gx) : s}\,\mathrm{Root}$$

Figure 4.8. Every school griped

school converts school, of type $np\backslash_n s$ as just described, to is a school, of type $np\backslash s$ like an ordinary verb phrase.) The new mode n is internal, so the direction of this slash is arbitrary, but we choose it for the notation to match the type $np\backslash s$ of an intransitive verb like slept.

A quantifier like every combines with a noun to the right to form a quantificational noun phrase. Because nouns have the type $np\backslash_n s$ and quantificational noun phrases have the type $s/\!\!/(np\backslash\!\!\backslash s)$, we let every take the type

$$(s/\!\!/(np\backslash\!\!\backslash s))/(np\backslash_n s). \tag{4.23}$$

Figure 4.8 straightforwardly derives the sentence every school griped. If we add the lexical items

$$\mathsf{dean\ of} \vdash (np\backslash_n s)/np, \tag{4.24}$$

$$\mathsf{Harvard} \vdash np, \tag{4.25}$$

then essentially the same derivation generates Every dean of Harvard griped.

How does the new binary mode n interact with the continuation mode $\circledcirc$? We broaden our notion of context to allow descending recursively into either branch of a constituent built with the n mode. To do so, we add the following rules alongside those in (4.10) on page 97.

$$\dfrac{\Gamma[\Theta \circledcirc (\Pi,_n \Delta)] \vdash T}{\Gamma[(\Theta,_n \Pi) \circledcirc \Delta] \vdash T}\,\mathrm{Left}_n \qquad \dfrac{\Gamma[(\Theta,_n \Pi) \circledcirc \Delta] \vdash T}{\Gamma[\Pi \circledcirc (\Delta,_n \Theta)] \vdash T}\,\mathrm{Right}_n \tag{4.26}$$

If we were to define contexts in the style of (4.21) and (4.22), these new rules would correspond respectively to

$$\dfrac{\Delta[\ ]}{\Delta[[\ ],_n \Pi]} \qquad \dfrac{\Delta[\ ]}{\Delta[\Theta,_n [\ ]]}. \tag{4.27}$$

$$\dfrac{\dfrac{a:\mathsf{a}\vdash a:(s/\!\!/(np\backslash\!\!\backslash s))/(np\backslash_n s)\qquad \dfrac{s:\mathsf{school}\vdash s:np\backslash_n s}{a:\mathsf{a},\,s:\mathsf{school}\vdash as:s/\!\!/(np\backslash\!\!\backslash s)}\,/\mathrm{E}\qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x:np\vdash x:np}\,\mathrm{Id}\qquad \dfrac{d:\mathsf{dean\ of}\vdash d:(np\backslash_n s)/np\qquad \dfrac{}{y:np\vdash y:np}\,\mathrm{Id}}{d:\mathsf{dean\ of},\,y:np\vdash dy:np\backslash_n s}\,/\mathrm{E}}{x:np,_n(d:\mathsf{dean\ of},\,y:np)\vdash dyx:s}\,\backslash_n\mathrm{E}}{(x:np,_n(d:\mathsf{dean\ of},\,y:np))\circledcirc 1\vdash dyx:s}\,\mathrm{Root}}{(d:\mathsf{dean\ of},\,y:np)\circledcirc(1,_n x:np)\vdash dyx:s}\,\mathrm{Right}_n}{y:np\circledcirc((1,_n x:np),d:\mathsf{dean\ of})\vdash dyx:s}\,\mathrm{Right}}{(1,_n x:np),d:\mathsf{dean\ of}\vdash\lambda y.dyx:np\backslash\!\!\backslash s}\,\backslash\!\!\backslash\mathrm{I}}{(a:\mathsf{a},\,s:\mathsf{school})\circledcirc((1,_n x:np),\,d:\mathsf{dean\ of})\vdash as(\lambda y.dyx):s}\,/\!\!/\mathrm{E}}{(d:\mathsf{dean\ of},\,(a:\mathsf{a},\,s:\mathsf{school}))\circledcirc(1,_n x:np)\vdash as(\lambda y.dyx):s}\,\mathrm{Right}}{(x:np,_n(d:\mathsf{dean\ of},\,(a:\mathsf{a},\,s:\mathsf{school})))\circledcirc 1\vdash as(\lambda y.dyx):s}\,\mathrm{Right}_n}{x:np,_n(d:\mathsf{dean\ of},\,(a:\mathsf{a},\,s:\mathsf{school}))\vdash as(\lambda y.dyx):s}\,\mathrm{Root}}{d:\mathsf{dean\ of},\,(a:\mathsf{a},\,s:\mathsf{school})\vdash\lambda x.as(\lambda y.dyx):np\backslash_n s}\,\backslash_n\mathrm{I}$$

Figure 4.9. Letting a school take narrow scope within dean of a school

A welcome consequence of this revision is that a scope ambiguity between two quantifiers is predicted even when one is located within the other's restrictor. This ambiguity is illustrated in the following sentence.

(4.28) Every dean of a school griped.

As Dalrymple et al. note (1999; Section 2.5.1), it can be tricky to account for linear scope in this sentence because there is no pronounced clause ("syntactic unit at the f-structure level") over which a school can take scope in the restrictor dean of a school. Nevertheless, the "imaginary clause" s in the type $np\backslash_n s$ for nouns suffices for a school to take scope over. Figure 4.9 proves that dean of a school has the type $np\backslash_n s$, just like a common noun. The Right$_n$ structural rule is crucial to this derivation. As the use of $/\!\!/$ E in Figure 4.9 indicates, a school takes scope entirely within the complex "noun" dean of a school, so inserting this proof into a derivation for (4.28) generates the linear-scope reading. The inverse-scope reading is also generated, without using either rule in (4.26). The top of that derivation proves

(4.29) $$(\mathsf{every},(\mathsf{dean\ of},np)),\mathsf{griped}\vdash s.$$

4.3.2. Islands. If we add a new mode of combination to a grammar without also adding any structural rule relating the new mode to the continuation mode $\circledcirc$, then contexts would be unable to cross the new mode. That is, the new mode would be an *island*.

For example, many linguists believe that a quantificational noun phrase cannot take scope outside of a *tensed clause*. A tensed clause is a clause whose main verb is inflected for tense. For example, the clause everyone left in the sentence

(4.30) Someone thought everyone left

is tensed, but the clause everyone to leave in the sentence

(4.31) Someone asked everyone to leave

is untensed. The usual way in type-logical grammar to make tensed clauses into scope islands is using an external unary mode i, whose unary connectives are $\Diamond_i$ and $\Box_i^{\downarrow}$. For instance, if the type of thought is $(\Box_i^{\downarrow}(np\backslash s))/s$, then Someone thought everyone left has only the linear-scope reading, on which someone takes wides scope over everyone. This is because

(4.32) $$np, (\mathsf{thought}, (np, \mathsf{left})) \vdash s$$

is not derivable—only

(4.33) $$np, \langle \mathsf{thought}, (np, \mathsf{left})\rangle_i \vdash s$$

is derivable, using the $\Box_i^{\downarrow}$ E rule as follows.

(4.34)
$$\dfrac{\dfrac{}{x : np \vdash x : np}\,\mathrm{Id} \quad \dfrac{\dfrac{t : \mathsf{thought} \vdash t : (\Box_i^{\downarrow}(np\backslash s))/s \quad \dfrac{\dfrac{}{y : np \vdash y : np}\,\mathrm{Id} \quad l : \mathsf{left} \vdash l : np\backslash s}{y : np, l : \mathsf{left} \vdash ly : s}\,\backslash\mathrm{E}}{t : \mathsf{thought}, (y : np, l : \mathsf{left}) \vdash t(ly) : \Box_i^{\downarrow}(np\backslash s)}\,/\mathrm{E}}{\langle t : \mathsf{thought}, (y : np, l : \mathsf{left})\rangle_i \vdash t(ly) : np\backslash s}\,\Box_i^{\downarrow}\mathrm{E}}{x : np, \langle t : \mathsf{thought}, (y : np, l : \mathsf{left})\rangle_i \vdash t(ly)x : s}\,\backslash\mathrm{E}$$

Without a structural rule relating the unary mode i to the binary mode $\circledcirc$, everyone in the embedded tensed clause cannot cross the i mode, which it must to access its entire surrounding context $np, \langle \mathsf{thought}, ([\,], \mathsf{left})\rangle_i$. In other words, thought everyone left is a scope island.

Another way to make tensed clauses be scope islands is to assign thought the type $(np\backslash s)/\Diamond_i s$, so that (4.32) is not derivable (nor is (4.33)) but

(4.35) $$np, (\mathsf{thought}, \langle np, \mathsf{left}\rangle_i) \vdash s$$

is derivable, using the $\Diamond_i$ I rule as follows.

(4.36)
$$\dfrac{\dfrac{}{x : np \vdash x : np}\,\mathrm{Id} \quad \dfrac{t : \mathsf{thought} \vdash t : (np\backslash s)/\Diamond_i s \quad \dfrac{\dfrac{\dfrac{}{y : np \vdash y : np}\,\mathrm{Id} \quad l : \mathsf{left} \vdash l : np\backslash s}{y : np, l : \mathsf{left} \vdash ly : s}\,\backslash\mathrm{E}}{\langle y : np, l : \mathsf{left}\rangle_i \vdash ly : \Diamond_i s}\,\Diamond_i\mathrm{I}}{t : \mathsf{thought}, \langle y : np, l : \mathsf{left}\rangle_i \vdash t(ly) : np\backslash s}\,/\mathrm{E}}{x : np, (t : \mathsf{thought}, \langle y : np, l : \mathsf{left}\rangle_i) \vdash t(ly)x : s}\,\backslash\mathrm{E}$$

Again, without a structural rule relating the unary mode i to the binary mode $\circledcirc$, everyone in the embedded tensed clause cannot access the context np, (thought, $\langle[\,], \mathsf{left}\rangle_i$). In other words, everyone left is a scope island.

Because we have only considered quantifiers that take scope over a clause (type s), it makes no observable difference whether the verb phrase thought everyone left or the clause everyone left is a scope island. If we have reason to treat an expression E as a quantifier that takes scope over a verb phrase (type $np\backslash s$) rather than a clause—perhaps E is an adverbial phrase of some sort—then we may be able to distinguish between the two kinds of scope islands by checking whether E can take scope over the entire verb phrase thought Alice E left. We do not pursue such a test here.

4.4. Linear order and evaluation order

The linear order of quantifiers in a sentence can affect its interpretation. For example, it is frequently observed that Mandarin quantifiers tend to take linear scope (Huang 1982; Aoun and Li 1993), and quantifier scope seems to be overtly expressed by syntactic raising in Hungarian (Szabolcsi 1997). In this section, we model quantifier order by restricting the notion of context, drawing from the restriction of evaluation contexts in Section 3.1.2 to specify evaluation order in a programming language. In Section 4.5 below, we apply the same concept of evaluation order to other linguistic side effects, namely anaphora, interrogation, and polarity sensitivity, which are also sensitive to linear order.

Figure 4.7 on page 102 successfully derives inverse scope for the sentence Someone saw everyone because everyone can take as argument its context

(4.37) $\qquad (1, \mathsf{someone}), \mathsf{saw},$

or equivalently,

(4.38) $\qquad \mathsf{someone}, (\mathsf{saw}, [\,]).$

In general, the scope of one quantifier contains another just in case the latter appears in the context argument of the former. Imagine for the moment that we are studying a language that resembles English but mandates linear scope. To rule out inverse scope, we can refine our notion of context so as to rule out any quantifier linearly located to the left of [ ]. Then (4.38), in which the quantifier someone occurs to the left of [ ], would no longer be a legitimate context, whereas

(4.39) $\qquad \mathsf{Alice}, (\mathsf{saw}, [\,])$

and

(4.40) $\qquad [\,], (\mathsf{saw}, \mathsf{everyone})$

would still be legitimate contexts. (The context (4.39) is used in Figure 4.3 on page 99 and Figure 4.4 on page 100 to derive Alice saw everyone. The

context (4.40) is used in (4.14) on page 99 and Figure 4.6 on page 101 to derive the linear-scope reading of Someone saw everyone.)

We implement this idea using a unary mode. Following terminology in the study of computational side effects, we call an expression *pure* if it contains no quantifier (that is, incurs no control effect), or *impure* otherwise. In particular, all values (in the sense of Section 3.1.2) are pure. To distinguish pure expressions from impure ones, we tag the types of pure expressions with a unary mode $\diamond$, or equivalently, the unary structural punctuation $\langle\ \rangle$. Any formula can be turned pure by embedding it under $\langle\ \rangle$ using the T rule.

$$\frac{\Gamma[\langle\Delta\rangle] \vdash T}{\Gamma[\Delta] \vdash T}\ \mathrm{T} \tag{4.41}$$

The T rule is analogous to *quotation* or *staging* in programming languages, which turns executable code into static data. Two quotations can be concatenated using the K′ rule.

$$\frac{\Gamma[\langle\Theta, \Pi\rangle] \vdash T}{\Gamma[\langle\Theta\rangle, \langle\Pi\rangle] \vdash T}\ \mathrm{K'} \tag{4.42}$$

Whereas the other rules introduced so far allow inference from top to bottom as well as from bottom to top, these rules only allow inference from top to bottom. We do not allow the converse of T because we want to regulate when inverse scope is available: we add a partial converse of T, called Unquote, in (4.47) on page 112 below. The converse of K′ appears harmless but is unneeded, so we leave it out.[5]

We consider a derivation complete if and only if it culminates in the type $\diamond s$, rather than s as before. The type $\diamond s$ signifies a pure clause rather than a quantifier over clauses (propositions). By contrast, everyone and someone are impure: their type $\diamond s /\!\!/ (np \backslash\!\!\backslash \diamond s)$ is not surrounded by $\diamond$, though the propositions they quantify over and finally produce are pure (hence the $\diamond$ in $\diamond s$).

To rule out inverse-scope contexts like (4.38), we modify our notion of context to require that the left branch of a type environment built up using the default mode be pure before descending recursively into the right branch. That is, we revise our notion of context from (4.21) on page 102 and (4.22) on page 103 to

$$\frac{}{[\ ]} \qquad \frac{\Delta[\ ]}{\Delta[[\ ], \Pi]} \qquad \frac{\Delta[\ ]}{\Delta[\langle\Theta\rangle, [\ ]]} \tag{4.43}$$

In terms of structural rules, we replace the Right rule from (4.10) on page 97 with

[5] Moot (2002; pages 39 and 156) explains these rules and their names. Curiously, the present work seems to be the first linguistic application of K′.

$$
\dfrac{\mathsf{someone} \vdash \Diamond s \mathbin{/\!\!/} (np \mathbin{\backslash\!\!\backslash} \Diamond s) \qquad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{everyone} \vdash \Diamond s \mathbin{/\!\!/} (np \mathbin{\backslash\!\!\backslash} \Diamond s) \qquad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots\ (4.14)\text{ on page 99}}{np, (\mathsf{saw}, np) \vdash s}}{\langle np, (\mathsf{saw}, np)\rangle \vdash \Diamond s}\,\Diamond\,\mathrm{I}}{\langle np\rangle, \langle \mathsf{saw}, np\rangle \vdash \Diamond s}\,\mathrm{K}'}{\langle np\rangle, (\langle \mathsf{saw}\rangle, \langle np\rangle) \vdash \Diamond s}\,\mathrm{K}'}{\langle np\rangle, (\langle \mathsf{saw}\rangle, np) \vdash \Diamond s}\,\mathrm{T}}{(\langle np\rangle, (\langle \mathsf{saw}\rangle, np)) \circledcirc 1 \vdash \Diamond s}\,\mathrm{Root}}{(\langle \mathsf{saw}\rangle, np) \circledcirc (1, \langle np\rangle) \vdash \Diamond s}\,\mathrm{Right}'}{np \circledcirc ((1, \langle np\rangle), \langle \mathsf{saw}\rangle) \vdash \Diamond s}\,\mathrm{Right}'}{(1, \langle np\rangle), \langle \mathsf{saw}\rangle \vdash np \mathbin{\backslash\!\!\backslash} \Diamond s}\,\backslash\!\!\backslash\,\mathrm{I}}
{\mathsf{everyone} \circledcirc ((1, \langle np\rangle), \langle \mathsf{saw}\rangle) \vdash \Diamond s}\,/\!\!/\,\mathrm{E}}{(\langle \mathsf{saw}\rangle, \mathsf{everyone}) \circledcirc (1, \langle np\rangle) \vdash \Diamond s}\,\mathrm{Right}'}{(\langle np\rangle, (\langle \mathsf{saw}\rangle, \mathsf{everyone})) \circledcirc 1 \vdash \Diamond s}\,\mathrm{Right}'}{(np, (\langle \mathsf{saw}\rangle, \mathsf{everyone})) \circledcirc 1 \vdash \Diamond s}\,\mathrm{T}}{(np, (\mathsf{saw}, \mathsf{everyone})) \circledcirc 1 \vdash \Diamond s}\,\mathrm{T}}{np \circledcirc ((\mathsf{saw}, \mathsf{everyone}) \circledcirc 1) \vdash \Diamond s}\,\mathrm{Left}}{(\mathsf{saw}, \mathsf{everyone}) \circledcirc 1 \vdash np \mathbin{\backslash\!\!\backslash} \Diamond s}\,\backslash\!\!\backslash\,\mathrm{I}}
{\dfrac{\dfrac{\mathsf{someone} \circledcirc ((\mathsf{saw}, \mathsf{everyone}) \circledcirc 1) \vdash \Diamond s}{(\mathsf{someone}, (\mathsf{saw}, \mathsf{everyone})) \circledcirc 1 \vdash \Diamond s}\,\mathrm{Left}}{\mathsf{someone}, (\mathsf{saw}, \mathsf{everyone}) \vdash \Diamond s}\,\mathrm{Root}}\,/\!\!/\,\mathrm{E}
$$

Figure 4.10. Linear scope for Someone saw everyone, under left-to-right evaluation. Bernardi (2002; page 50) shows the $\Diamond$ I rule used in this derivation (same as $\Diamond$ R in the Gentzen presentation (Moortgat 1997; Definition 4.16)), as well as the other natural-deduction rules for the unary operators $\Diamond$ and $\Box^{\downarrow}$, namely $\Diamond$ E, $\Box^{\downarrow}$ I (same as $\Box^{\downarrow}$ R), and $\Box^{\downarrow}$ E.

a more specific instance Right′.

(4.44)
$$
\frac{\Gamma[\Theta \circledcirc (\Pi, \Delta)] \vdash T}{\Gamma[(\Theta, \Pi) \circledcirc \Delta] \vdash T}\ \text{Left} \qquad \frac{\Gamma[(\langle\Theta\rangle, \Pi) \circledcirc \Delta] \vdash T}{\Gamma[\Pi \circledcirc (\Delta, \langle\Theta\rangle)] \vdash T}\ \text{Right}'
$$

With these changes to the grammar, only the linear-scope reading for Someone saw everyone (of type $\Diamond s$) remains derivable. Figure 4.10 shows the derivation. It is shaped exactly like Figure 4.6 on page 101, except the T, K′, and $\Diamond$ I

rules are used after (4.14) on page 99 to prove

(4.45) $\langle np\rangle, (\langle \mathsf{saw}\rangle, np) \vdash \Diamond s.$

The inverse-scope derivation in Figure 4.7 on page 102 is no longer valid, because someone is impure and so the Right′ rule in (4.44) does not apply.

What we have just seen is that a linguistic preference for linear scope reflects a computational preference for left-to-right evaluation. Of course, in many languages (including English), inverse scope is available. That does not mean that order-of-evaluation effects are absent—it only means that, if they are present, they are more subtle. In the rest of this section, we examine one way to reintroduce inverse scope that accounts for additional empirical observations in Section 4.5 below.

The basic idea is to treat inverse scope as *multistage programming* (see Calcagno et al. 2003, Taha and Nielsen 2003, and references therein). A multi-stage program is a program that generates another program to be run later. The generating program is said to execute in an *earlier* or *outer* stage, and the generated program is said to execute in a *later* or *inner* stage. More than two stages are also possible. Evaluation in each stage is ordered separately: later-stage code runs only after earlier-stage code generates it. Informally, then, we can treat the inverse-scope reading of Someone saw everyone as the following outer-stage program.

(4.46) Run the program consisting of the word someone, the word saw, and everyone.

Serifs are significant here: The sentence above only uses one quantifier (everyone). It also mentions a word (someone) that is a quantifier, but does not use it. If the domain of people under discussion is Alice, Bob, and Carol, then (4.46) conjoins the meanings of the sentences Someone saw Alice, Someone saw Bob, and Someone saw Carol. These three sentences are the inner-stage programs.

A typical multistage programming language provides facilities for:

- creating programs (by quotation or staging);
- combining programs (by concatenation); and
- running programs (by *unquotation* or evaluation).[6]

Intuitively, a quoted value is an inactive piece of program text that does not execute until the "wrapping" $\Diamond$ (or equivalently $\langle\ \rangle$) is removed. Removing the wrapping turns inactive data into active code.

[6]We call this step "unquotation" despite the fact that "evaluation" or "eval" is the commonly used term for the same concept in the staged programming literature, to avoid confusion with the concept of evaluation order just discussed.

The T and K′ rules in (4.41) and (4.42) are our facilities for quoting and concatenating programs, respectively. To run programs, we add the Unquote rule.

$$\frac{\Gamma[\langle\Delta\rangle_u] \vdash T}{\Gamma[\langle\langle\Delta\rangle_u\rangle] \vdash T}\ \text{Unquote} \tag{4.47}$$

Because the type environment of a quoted program is enclosed in $\langle\ \rangle$, to run a program—to unquote it—is to remove that $\langle\ \rangle$. Although quotation applies freely (using T), unquotation does not. Rather, the Unquote rule above only unquotes a type environment of the form $\langle\Delta\rangle_u$. Here u is a unary mode that marks those types that are allowed as the answer type of a quoted program; this mode can be thought of as a feature (in the linguistic sense) checked by the Unquote rule. In particular, we hereafter take the clause type s to be shorthand for $\diamond_u s'$ (or $\langle s'\rangle_u$ in a type environment), where s' is an atomic formula distinct from s, so that we can derive s from the quoted clause type $\diamond s$ using the Unquote rule.[7]

Figure 4.11 shows how to derive the inverse-scope reading of Someone saw everyone once again using Unquote. (Because s and $\diamond s$ entail each other in the presence of Unquote, we can restore the lexical types of someone and everyone from $\diamond s /\!\!/ (np \backslash\!\!\backslash \diamond s)$ back to $s /\!\!/ (np \backslash\!\!\backslash s)$.) As noted above, the Unquote rule is necessary to derive this reading using the Right′ rule in (4.44) on page 110. More precisely, in order for one quantifier to take inverse scope over another, the Unquote rule must apply to the clause produced by the narrower-scope quantifier (here someone).

Our application of multistage programming to natural language is a new contribution that builds on the idea of evaluation order. The unary modes $\diamond$ and $\diamond_u$, which we use to encode this application, add considerable complexity to the basic analysis of Someone saw everyone given above in Section 4.2. The payoff, in the next section, is to account for fairly subtle and complex linguistic phenomena by controlling evaluation order. In programming-language theory, modal logic has also been used in type systems for quotation and staging (Davies and Pfenning 1996; Davies 1996; Davies and Pfenning 2001; Goubault-Larrecq 1996a,b,c, 1997). It is unclear whether the uses of unary modes there and here are related. In future work, we hope to investigate this possible relationship and, perhaps through it, explain the structural rules T, K′, and Unquote more systematically.

4.5. Beyond quantification

In this section, we survey continuation-based analyses of several linguistic side effects and transliterate them into type-logical grammar. The phenomena

[7] In this paper, s is the only type that can be unquoted, that is, the only type of the form $\diamond_u A$. A treatment of polarity sensitivity that is less simplistic than the one in Section 4.5.3 calls for multiple clause types that can be unquoted (Shan 2004b).

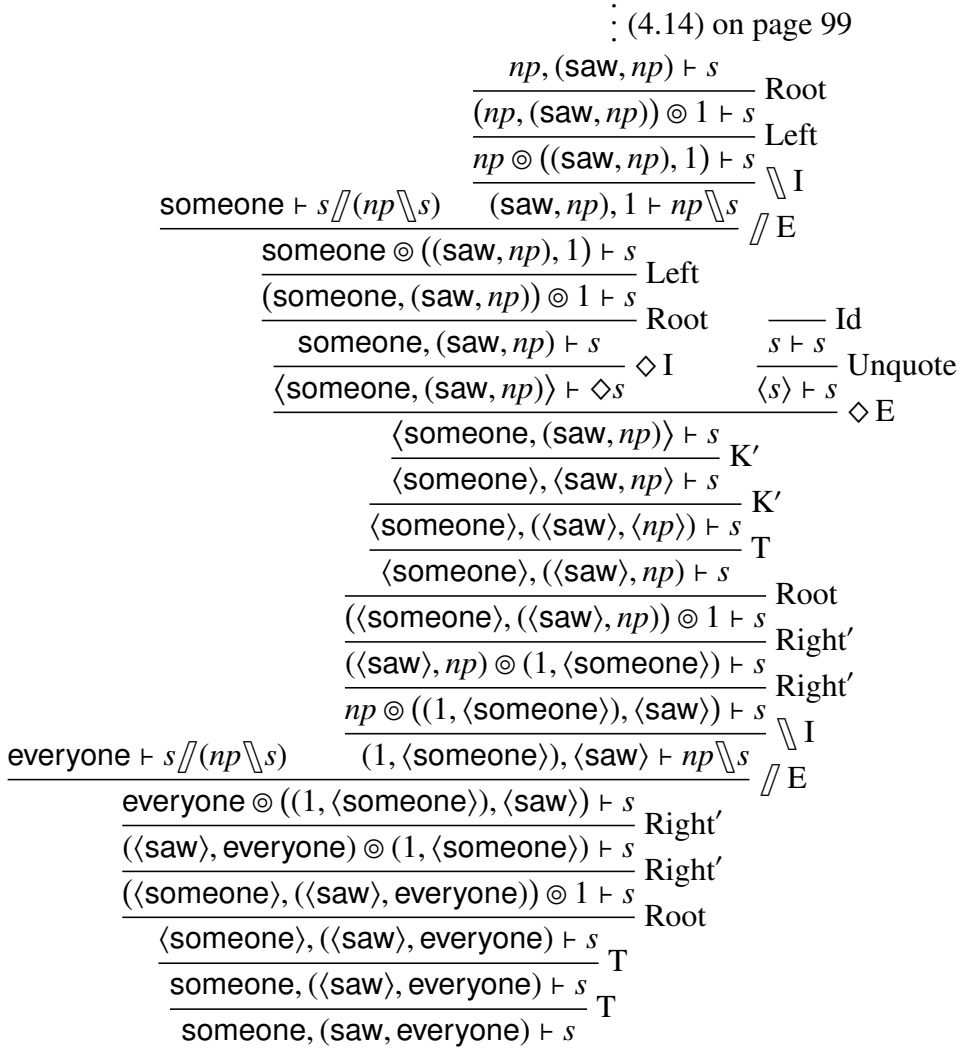


Figure 4.11. Inverse scope for Someone saw everyone, under left-to-right evaluation, using unquotation

discussed are quite intricate, and we cannot hope to be comprehensive here—the papers cited provide more details. Our point is that continuation-based analyses account for more empirical observations and with more theoretical unity than previous treatments of the same phenomena in type-logical grammar.

4.5.1. Anaphora and crossover. As alluded to in Section 1.3.1, we follow the view of *dynamic semantics* (Groenendijk and Stokhof 1991; Heim 1982; Kamp 1981) that anaphora is the storage and retrieval of discourse referents,

analogous to the computational side effect of state. Under this view, anaphora is an ideal arena in which to test the utility of continuations and our side-effect analogy in linguistics, for two reasons. First, continuations model side effects, including state, a prototypical computational side effect (Filinski 1994; Section 5.4). Second, continuations model how multiple side effects interact, and anaphora interacts with quantification and interrogation.

Shan and Barker (2005) argue that evaluation order explains two distinct phenomena in English, *crossover* and *superiority*. Here we express that analysis in type-logical grammar and compare our approach to Jäger's (2001) analysis of crossover. We account for more empirical cases where crossover interacts with *wh-raising*. We also argue that crossover in the linguistic side effect of anaphora and superiority in the linguistic side effect of interrogation stem from the same underlying mechanism, namely, left-to-right evaluation as implemented in Section 4.4.

Crossover is the name for the fact that, if a quantificational noun phrase serves as the antecedent of a pronoun, the antecedent usually must precede the pronoun (Postal 1971).

(4.48) a. Everyone$_i$ saw his$_i$ mother.
b. *His$_i$ mother saw everyone$_i$.

We use standard linguistics notation above to indicate the relationship between a pronoun and its antecedent: by attaching the same subscript (such as $_i$) to both. Thus (4.48) indicates that the sentence (4.48a) can correspond to the logical formula

(4.49) $$\forall x.\, \mathrm{saw}(x, \mathrm{mother}(x)),$$

but (4.48b) cannot correspond to

(4.50) $$\forall x.\, \mathrm{saw}(\mathrm{mother}(x), x).$$

When anaphora succeeds, the antecedent is said to *bind* the pronoun. We derive crossover by assuming that people evaluate expressions from left to right by default, and that a quantifier must be evaluated before any pronoun it binds.

To explain crossover, we must first implement anaphora in our grammar. Many implementations of anaphora exist for type-logical grammar and related approaches (Szabolcsi 1989, 1992; Dowty 1992; Moortgat 1996; Morrill 2000; Hepple 1990, 1992; Jacobson 1999, 2000). Jäger (2001) surveys these implementations. He then proposes a new logical connective "|" with special inference rules. Pronouns have type $np|np$; more generally, $T_2|T_1$ means "I behave like something of type T_2, but need to be bound by something of type T_1". Jäger's inference rules allow any expression of type T_1 to bind something of type $T_2|T_1$ as long as the antecedent precedes the pronoun.

Two features of Jäger's proposal are most relevant to us: resource duplication and linear order.

Resource duplication: In standard type-logical grammar, every linguistic resource is used exactly once, never duplicated. By contrast, anaphora uses a resource twice: once for the antecedent and then again for the bound pronoun. For example, the variable x in (4.49) is used twice: once as the seer and once as the offspring. To deal with this mismatch, the inference rules for | incorporate a limited[8] form of resource duplication.

Linear order: Like us but unlike in most accounts of anaphora based on Logical Form (May 1985; Reinhart 1983), Jäger recognizes the role of linear order. His system requires that the antecedent precede the pronoun it binds, and he gives empirical arguments that anaphora is sensitive to linear order. In particular, his system correctly predicts crossover facts like those in (4.48).

In part because Jäger is primarily interested in resource duplication, he merely stipulates linear order: it follows from nothing, and the rules could just as easily have required that the antecedent follow the pronoun it binds. We explain the role of linear order in anaphora in a deeper way: Our account rules out crossover by assuming that people evaluate expressions from left to right by default.

To help compare our approach with Jäger's (2001), we also accomplish anaphora by means of a (single) inference rule that incorporates a limited form of resource duplication. However, whereas Jäger extends type-logical grammar with a new logical connective |, we express binding relationships with a new binary mode b (mnemonic for "binding"). For example, the pronoun she has the lexical entry

$$\textsf{she} \vdash (np\backslash_b T)/\!\!/(np\backslash\!\!\backslash T). \tag{4.51}$$

Conceptually, $np\backslash_b T$ is the type of an expression that would otherwise count as a T, but which contains a pronoun waiting to be bound by an np. The type for she indicates that it behaves locally like an np, but turns an enclosing expression of type T into something of type $np\backslash_b T$. For instance, Bob thought she left would have type $np\backslash_b s$.

[8] Jäger details logical and linguistic reasons to limit resource duplication. Logically, Jäger's system preserves the finite-reading property alongside cut elimination, decidability, and the subformula property. Linguistically, the sentence The claim$_i$ that someone saw everyone and its$_i$ negation are both true cannot possibly be true; to rule out the spurious reading where someone saw everyone takes, say, linear scope in the antecedent but inverse scope in the bound pronoun it, we must only duplicate formulas like np, not someone, (saw, everyone). Thus the | connective must not obey contravariance: $A \vdash B$ must not entail $np|B \vdash np|A$.

Our binding rule says that a subexpression *np* within a context Δ[ ] can bind into (that is, bind a pronoun inside) the larger expression Δ[*np*].

$$\frac{\Gamma[np,_b (np \circledcirc \Delta)] \vdash T}{\Gamma[np \circledcirc \Delta] \vdash T}\ \text{Bind} \tag{4.52}$$

This rule duplicates an *np* resource:[9] it says that, whenever two *np*s in $np,_b (np \circledcirc \Delta)$ suffice to derive T, a single *np* in $np \circledcirc \Delta$ will do. In the former type environment $np,_b (np \circledcirc \Delta)$, the second *np* is an antecedent in the context Δ, and the first *np* is a copy of the antecedent that provides the bound meaning to the pronoun. Read bottom-up, the Bind rule says that an *np* in a context Δ can be duplicated to bind into Δ[*np*].

To illustrate, Figure 4.12 derives Everyone$_i$ saw his$_i$ mother, in which the subexpression everyone binds into the context [ ] saw his mother. On our analysis, the pronoun itself takes scope over its antecedent's context. In Figure 4.12, everyone binds he, and the context of everyone is [ ] saw his mother. (This context appears at the bottom of the derivation, where everyone enters, as (saw, (he, 's mother)), 1.) Thus he must take scope over this context—that is, must take as *its* context *np* saw [ ]'s mother. (This context appears in the middle of the derivation, where he enters, as 's mother, $((1, \langle np \rangle), \langle \text{saw} \rangle)$.)

It is always easier to explain why a good derivation works than why a bad sentence has no derivation, but let us offer a few words on why the crossover example *His$_i$ mother saw everyone$_i$ (4.48b) does not have the reading indicated by the subscripts. The crucial difference between (4.48a) and (4.48b) is that everyone is now to the right of the pronoun it is trying to bind. In order for everyone to take scope over the context his mother saw [ ], we must apply the Right′ rule twice (just as in the inverse-scope derivation above in Figure 4.7 on page 102). But the Right′ rule requires quoting his mother and saw with two ◇s. Once ◇s enter the picture, the only way to remove them is the Unquote rule. But since Unquote only applies to expressions of type *s*, and there is no way to arrive at the type *s* without first unquoting the pronoun, there is an unresolvable standoff: once the non-unquotable $np\backslash_b s$ gets quoted, the derivation is doomed.

[9] Like Dalrymple et al. (1999), we limit resource duplication to *np* only—Footnote 8 explains why. This form of limited resource duplication is much more primitive than Jäger's, but suffices here because we focus on the role of linear order in anaphora and in other linguistic side effects that do not duplicate resources. It would be nice to combine Jäger's treatment of resource duplication with our treatment of linear order, because the former unifies noun-phrase anaphora with verb-phrase ellipsis while the latter unifies crossover with superiority, but we leave that to future work. One promising bridge between the two treatments is Morrill's account of anaphora in terms of discontinuous constituency (2000, 2003): perhaps ⊚ is associative in nature with 1 as its left as well as right identity, and Jäger's |L and |R rules can be rephrased as follows.

$$\frac{Y \vdash B \quad \Gamma[A \circledcirc B] \vdash C}{\Gamma[(A|B) \circledcirc Y] \vdash C}\ |\text{L} \qquad \frac{A_n \circledcirc \cdots \circledcirc A_1 \circledcirc K \vdash B}{(A_n|C) \circledcirc \cdots \circledcirc (A_1|C) \circledcirc K \vdash B|C}\ |\text{R}$$

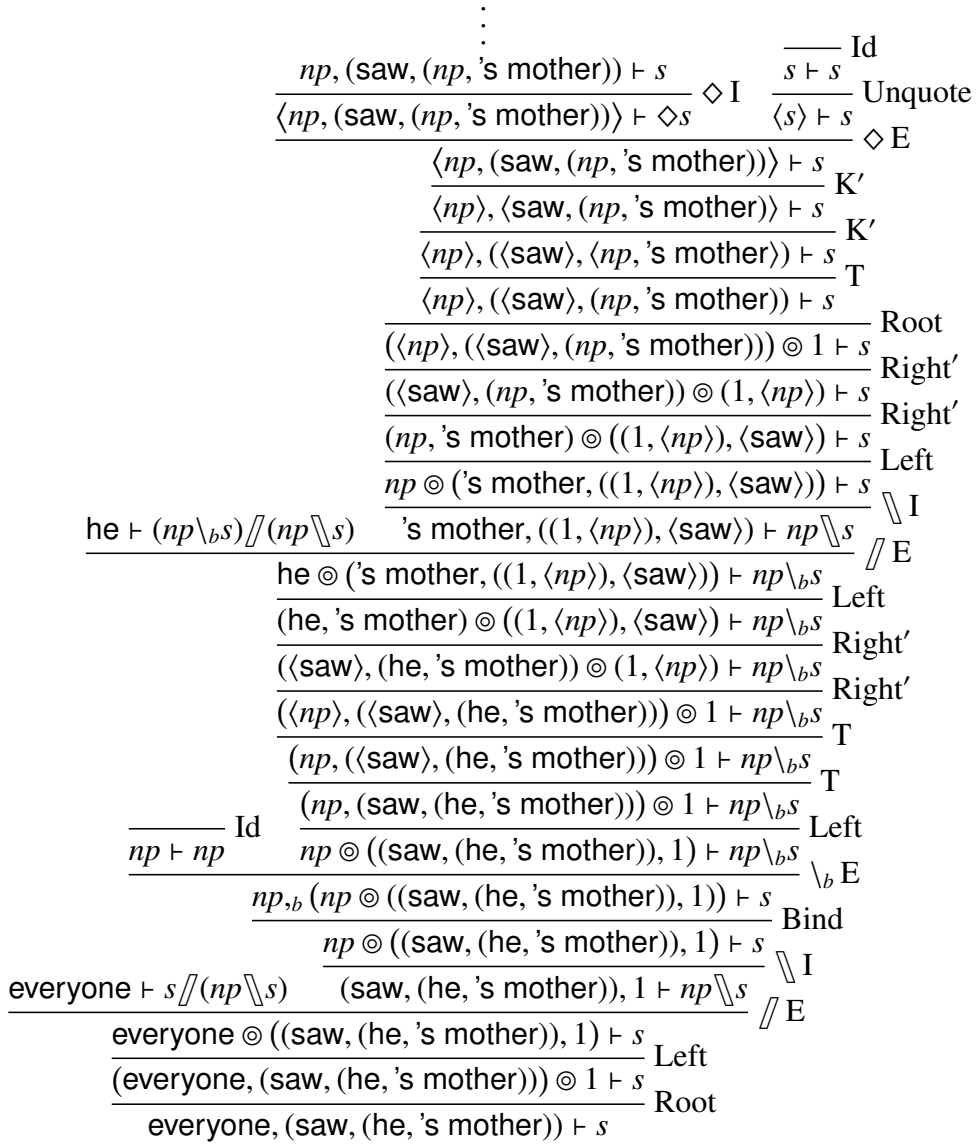


Figure 4.12. Everyone$_i$ saw his$_i$ mother

It is controversial whether the crossover constraint regulates anaphora according to linear order or according to *c-command* relations between the quantificational antecedent and the pronoun. We agree with Jäger that linear order is the determining factor, contra Reinhart (1983). Thus we need to say nothing more for our theory to deal with a sentence like Everyone's$_i$ mother saw him$_i$, in which the antecedent everyone precedes but does not c-command the pronoun

him (cf. Büring 2001, 2004). One way in which we address Reinhart's objections to linear order is to distinguish linear order from evaluation order: Our claim is that anaphora depends on evaluation order, which is based on linear order but may differ from linear order due to certain syntactic constructions. For example, the *pied binding* examples in Section 4.5.2 below illustrate that a raised-wh question delays the evaluation of the raised wh-phrase. We discuss the role of c-command, linear order, and evaluation order in more detail elsewhere (Shan and Barker 2005; Sections 1.3 and 1.4).

4.5.2. Questions and superiority. We now turn to the linguistic side effect of interrogation, in other words, questions. Given that (as explained in Section 1.1) we measure the success of a natural-language semantic theory by how well it models when an utterance is true, and a question seems neither true nor false, it may be surprising that we consider questions at all. Indeed, questions are one reason why we may need other ways to appraise a semantic theory, such as whether it is felicitous to respond to a given question with a given answer. Questions are also an area of natural language where the boundary between semantics (meaning) and pragmatics (use) is less clear. Nevertheless, we can embed a question in a larger utterance that is true or false to learn indirectly what questions mean. For example, the wh-questions in (4.53) must have different meanings in a sound and compositional semantics, because the sentences in (4.54) may not be all true or all false.

(4.53) a. who saw Bob
b. who saw Carol
c. who saw what

(4.54) a. Alice knows who saw Bob.
b. Alice knows who saw Carol.
c. Alice knows who saw what.

Motivated by these examples, we restrict our attention in this section to embedded wh-questions, such as those in (4.53). The linguistic generalizations and analyses we discuss carry over to unembedded wh-questions, with minor adjustments to deal with subject-auxiliary inversion and verb forms.

For concreteness, we follow the *structured meaning* approach (Krifka 2001) and let a question (such as (4.53a)) denote a function from a short answer (such as Alice) to a proposition (such as that Alice saw Bob).[10] If the proper noun Bob denotes the individual b, and the transitive verb saw denotes the two-place function f, then we let the single-wh question (4.53a) denote the one-place

[10]Despite this choice of approach, our basic idea transfers to other approaches to question denotations (Hamblin 1973; Karttunen 1977; Groenendijk and Stokhof 1984, 1997; Nelken and Francez 2000, 2002; Nelken and Shan 2004).

function $\lambda x.\, fbx$. We also let the double-wh question (4.53c) denote the two-place function $\lambda x.\, \lambda y.\, fyx$, in which the variable x corresponds to the wh-word who, and the variable y to the wh-word what.

Just as we introduced a new binary mode b in Section 4.5.1 to express anaphora in syntax, we add a new binary mode ? here to express interrogation in syntax. Conceptually, if S is a type, then $np\backslash_? S$ is the type of a question function that maps a short answer of type np to a result of type S. For example, a single-wh question such as (4.53a) has the type $np\backslash_? s$, and a double-wh question such as (4.53c) has the type $np\backslash_? np\backslash_? s$.

As mentioned above, we explain *superiority* just as we explain crossover, using left-to-right evaluation order. Superiority is the name (Chomsky 1973) for the fact (Kuno and Robinson 1972) that a *wh-trace* usually must precede any additional *in-situ wh-phrase*. For example, the acceptable question (4.55b) (as in Alice knows who you think saw what) begins with the *raised wh-phrase* who. As is typical, this raised wh-phrase precedes the gap indicated by __, called a trace. The trace is associated with the raised wh-phrase in the sense that who asks for the seer. After the trace comes the in-situ wh-phrase what. In the unacceptable questions (4.55c) and (4.55d), the in-situ wh-phrase who precedes the trace.[11]

(4.55) a. who saw what
b. who you think __ saw what
c. *what who saw __
d. *what you think who saw __

We suggest that the in-situ wh-phrase who in (4.55c) and (4.55d) prevents the raised wh-phrase what from associating with its trace because a raised wh-phrase can associate with a trace only if the trace takes widest scope over any additional in-situ wh-phrase. As with crossover in anaphora, we appeal to linear order: we show that a wh-trace can take scope over an in-situ wh-phrase only if the trace comes first.

Of course, before we can explain superiority, we have to treat questions first. A *raised-wh* question is a wh-question like (4.56), in which a wh-phrase raises to the front and leaves behind a trace.

(4.56) whose mother Alice saw __

In this example, not only does the wh-word whose raise to the front, but the larger phrase whose mother raises together (technically, gets *pied-piped*) to the front.

[11] An unembedded question can have a so-called *echo-question* reading, which we exclude from consideration. For example, despite the unacceptability indicated in (4.55c), the question What did who see __? is perfectly acceptable in response to the question What did Alice see __?, if the speaker of the first question does not know who Alice is or did not hear the word Alice.

Less standard in English is an *in-situ-wh* question: a wh-question like (4.57), in which the wh-phrase stays in situ.

(4.57) Alice saw whose mother

We now provide lexical entries for who and what. (We analyze the phrase whose mother as who followed by 's mother.)

(4.58) $\mathsf{who} \vdash (np\backslash_? S) /\!\!/ (np \backslash\!\!\backslash S)$

(4.59) $\mathsf{what} \vdash (np\backslash_? S) /\!\!/ (np \backslash\!\!\backslash S)$

Here the type variable S ranges over all types, not just s, to accommodate multiple-wh questions. The lexical entries above specify that wh-words take scope in situ, so they by themselves generate in-situ-wh questions. To generate raised-wh questions, we add two structural rules.[12]

(4.60) $$\frac{\Gamma[\mathrm{A}, _? (\Theta \circledcirc (\Pi, \Delta))] \vdash T}{\Gamma[\mathrm{A}, _? (\Theta, (\Pi \circledcirc \Delta))] \vdash T}\ \text{Trace Left}$$

(4.61) $$\frac{\Gamma[\mathrm{A}, _? (\Pi \circledcirc (\Delta, \langle\Theta\rangle))] \vdash T}{\Gamma[\mathrm{A}, _? (\Pi, (\langle\Theta\rangle \circledcirc \Delta))] \vdash T}\ \text{Trace Right}'$$

Figure 4.13 shows the basic usage scenario for these additions to the grammar. First, the $/\!\!/$ E inference near the top concludes with the in-situ-wh question (4.57). Then, the entire derivation culminates in the raised-wh question (4.56).

Let us now examine the derivation of a double-wh question that abides by superiority, so that we can see what goes wrong in an attempt to violate superiority. Figure 4.14 derives the question who saw what in (4.55a), in which who is raised and what remains in situ. (We say that who is raised, even though raising who does not affect the sequence of words, because this derivation uses the Trace Left rule near the end. We exemplify raised-wh questions with who saw what for simplicity.) In Figure 4.14, Trace Left applies to the raised wh-phrase who in the context [ ] saw what. In other words, the raised wh-phrase takes scope over the rest of the sentence. Similarly, to derive the superiority violation *what who saw in (4.55c), the raised wh-phrase what must take scope over the context who saw [ ]. Just as in the previous section, taking scope over such a context requires using the Right′ rule twice, which introduces two $\diamond$s and necessitates Unquote.

[12] These two rules are really the composition of the Left and Right′ rules with a single new rule

$$\frac{\Gamma[\mathrm{A}, _? (\Theta \circledcirc \Delta)] \vdash T}{\Gamma[\mathrm{A}, _? (\Theta, (\_\_ \circledcirc \Delta))] \vdash T}\ \text{Trace},$$

where __ is the empty string, in other words the identity for the default mode. But the default mode in standard type-logical grammar does not have an identity. If multimodal type-logical grammar were to allow identities for all modes and use unary modes to enforce non-emptiness when necessary, then we would be able to replace Trace Left and Trace Right′ with this Trace rule—an appealing prospect.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np \vdash np}\,\mathrm{Id} \quad \dfrac{\mathsf{who} \vdash (np\backslash_? s)/\!\!/(np\backslash\!\!\backslash s) \;\; \text{'s mother}, ((1, \langle \mathsf{Alice}\rangle), \langle \mathsf{saw}\rangle) \vdash np\backslash\!\!\backslash s}{\mathsf{who} \circledcirc (\text{'s mother}, ((1, \langle \mathsf{Alice}\rangle), \langle \mathsf{saw}\rangle)) \vdash np\backslash_? s}\,/\!\!/\mathrm{E}}{np,_? (\mathsf{who} \circledcirc (\text{'s mother}, ((1, \langle \mathsf{Alice}\rangle), \langle \mathsf{saw}\rangle))) \vdash s}\,\backslash_?\mathrm{E}}{np,_? ((\mathsf{who}, \text{'s mother}) \circledcirc ((1, \langle \mathsf{Alice}\rangle), \langle \mathsf{saw}\rangle)) \vdash s}\,\mathrm{Left}}{np,_? ((\mathsf{who}, \text{'s mother}), (\langle \mathsf{saw}\rangle \circledcirc (1, \langle \mathsf{Alice}\rangle))) \vdash s}\,\mathrm{Trace\ Right'}}{(\mathsf{who}, \text{'s mother}), (\langle \mathsf{saw}\rangle \circledcirc (1, \langle \mathsf{Alice}\rangle)) \vdash np\backslash_? s}\,\backslash_?\mathrm{I}}{(\mathsf{who}, \text{'s mother}), ((\langle \mathsf{Alice}\rangle, \langle \mathsf{saw}\rangle) \circledcirc 1) \vdash np\backslash_? s}\,\mathrm{Right'}}{(\mathsf{who}, \text{'s mother}), (\langle \mathsf{Alice}\rangle, \langle \mathsf{saw}\rangle) \vdash np\backslash_? s}\,\mathrm{Root}}{(\mathsf{who}, \text{'s mother}), (\mathsf{Alice}, \langle \mathsf{saw}\rangle) \vdash np\backslash_? s}\,\mathrm{T}}{(\mathsf{who}, \text{'s mother}), (\mathsf{Alice}, \mathsf{saw}) \vdash np\backslash_? s}\,\mathrm{T}$$

⋮ like the top of Figure 4.12
⋮ on page 117

Figure 4.13. Deriving whose mother Alice saw from Alice saw whose mother

But questions cannot be unquoted: their types are of the form $A\backslash_? S$, not $\diamond_u S$. Superiority is thus enforced.

For Jäger, a raised wh-phrase has the type $q/(np \uparrow s)$, where q is the type of questions and $np \uparrow s$ is the type of an s with an np-trace. Because the mechanism that associates a raised wh-phrase with its trace is separate from anaphora, whatever might explain superiority in Jäger's system will be independent of his explanation for crossover. Our system uses continuations for anaphora as well as to associate a raised wh-phrase with its trace. Hence evaluation order explains crossover as well as why an intervening in-situ wh-phrase prevents a raised wh-phrase from associating with its trace.

It is debatable whether we should account for crossover and superiority with a unified explanation. Further investigation may reveal empirical arguments that these two phenomena are in fact distinct. One empirical observation that favors our unified treatment is what we call *pied binding*, a pattern of interaction between wh-raising and crossover. To start with, the typical raised wh-phrase can bind a pronoun only if the trace of the wh-phrase precedes the pronoun.

(4.62) who_i __ saw his_i mother

(4.63) *who_i his_i mother saw __

One popular explanation since Reinhart's work (1983) is that it is the trace that binds the pronoun. In Jäger's system (his pages 124–125), the trace can indeed bind the pronoun in examples such as (4.62). Jäger's system also explains why

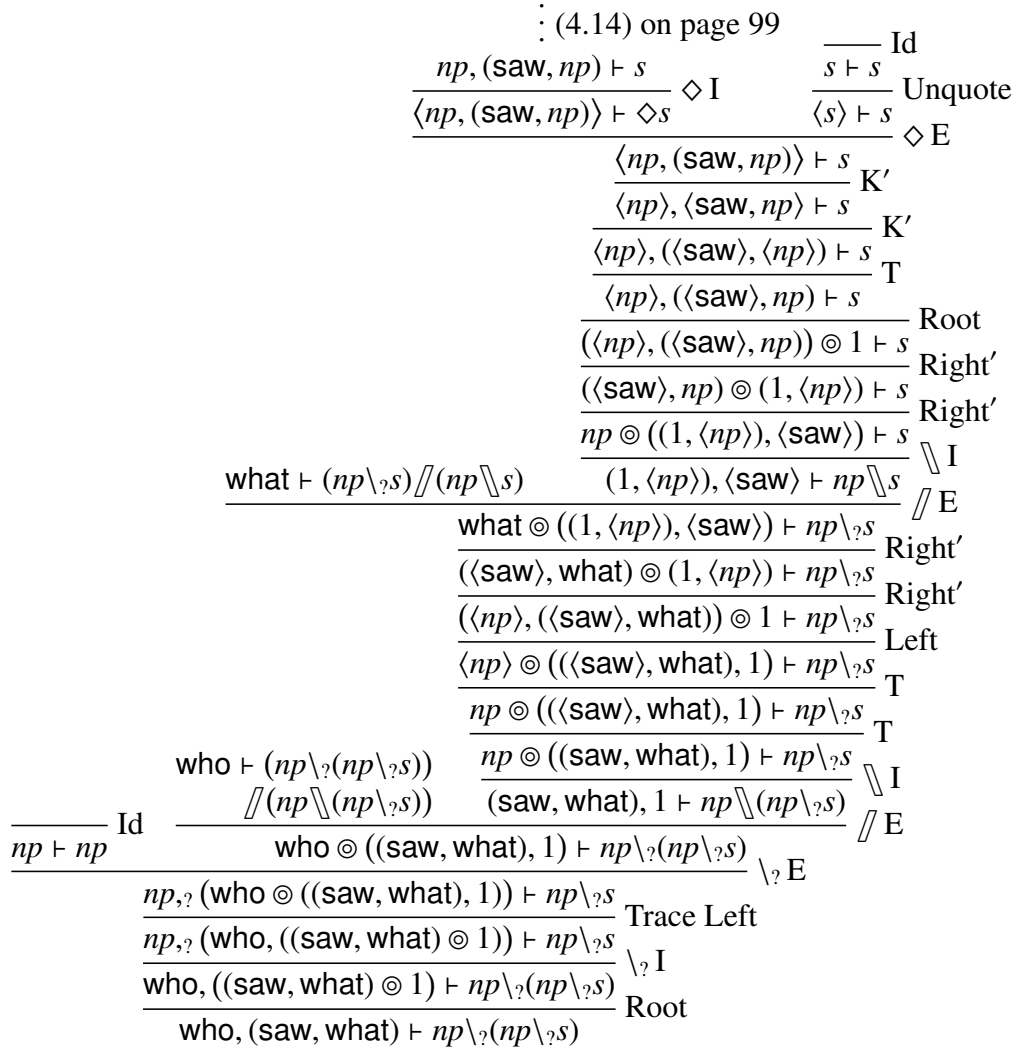


Figure 4.14. Who saw what

(4.63) is unacceptable: when the trace follows the pronoun, crossover rules out (4.63).

However, more complex examples cast doubt on the hypothesis that it is the trace that binds the pronoun.

(4.64) whose$_i$ father Alice said __ saw his$_i$ mother

(4.65) *whose$_i$ father Alice said his$_i$ mother saw __

In (4.64) and (4.65), the trace is associated with the entire pied-piped wh-phrase

whose father. That is, it is a father who is seeing or being seen. Yet, as the subscripts in (4.64) indicate, the wh-word who alone can bind the pronoun. This example suggests that the wh-word must be able to bind the pronoun directly after all.

Interestingly, the second example (4.65) shows that wh-binding still requires linear precedence of a sort: the wh-word who can bind the pronoun only if the trace precedes the pronoun. In our system, a wh-phrase can bind a pronoun just as a quantificational noun phrase does. However, just as it is a superiority violation for an in-situ wh-phrase to interfere between a raised wh-phrase and its trace, it is a crossover violation for a pronoun to interfere between an antecedent raised wh-phrase and its trace. Put differently, the raised-wh question (4.65) violates crossover because the corresponding in-situ-wh question Alice said his$_i$ mother saw whose$_i$ father violates crossover. In computational terms, the evaluation of a raised wh-phrase is delayed until its corresponding wh-trace. This complex pattern of interaction between wh-phrases and pronouns falls out from our unified account without additional stipulation.

To recap, our analysis of anaphora and interrogation in terms of in-situ quantification holds its own against Jäger's robust previous analysis: we account for both crossover and superiority, including the interaction between wh-raising and crossover. Before us, many linguists have also noted and tried to explain that the relation between an antecedent and a pronoun that violate crossover is similar to the relation between a wh-trace and an in-situ wh-phrase that violate superiority. Hornstein (1995) and Dayal (1996) separately argue that superiority reduces to crossover, but they rely on encoding multiple-wh questions as *functional-wh* questions (Chierchia 1991, 1993). O'Neil (1993) stipulates crossover and superiority as two parts of his Generalized Scope Marking Condition, but he does not unify them. By contrast, we unify crossover and superiority to a common hypothesis, namely left-to-right evaluation. This common hypothesis applies across linguistic side effects: not just to crossover in anaphora and superiority in interrogation, but also to the preference for linear scope in quantification, as explained in Section 4.4, and the effect of linear order on polarity sensitivity, which we turn to next.

4.5.3. Polarity sensitivity. The sentences (4.66) and (4.67) have the same truth conditions: knowing whether anyone slept and knowing whether someone slept are both knowing whether there exists an x such that x slept.

(4.66) Alice knows whether anyone slept.

(4.67) Alice knows whether someone slept.

Examples like these make us want to analyze anyone as an existential quantifier just like someone. Unfortunately for the simplicity of our theory, anyone and

someone (and more generally, any and some) are not always interchangeable in their existential usage. For example, whereas the sentence (4.68a) only has an inverse-scope reading $\exists y.\, \forall x.\, \neg \mathrm{saw}(x, y)$ (following Section 3.3.1's use of logical formulas to indicate meaning), the sentence (4.69a) only has a linear-scope reading $\forall x.\, \neg \exists y.\, \mathrm{saw}(x, y)$. Moreover, whereas the sentences (4.68b) and (4.68c) are acceptable, their counterparts (4.69b) and (4.69c) with someone replaced by anyone are unacceptable.

(4.68) a. Nobody saw someone.
b. Everyone saw someone.
c. Alice saw someone.

(4.69) a. Nobody saw anyone.
b. *Everyone saw anyone.
c. *Alice saw anyone.

These differences are termed *polarity sensitivity*, or *polarity licensing*, for the following reason (Ladusaw 1979; Krifka 1995; inter alia). The word anyone is a *negative polarity item*: to a first approximation, it can occur only in a *downward-entailing* context, such as in the scope of a *monotonically decreasing* quantifier. (Dually, the word someone is a *positive polarity item*, but we focus on anyone here.) A quantifier q, of type $(np \to s) \to s$, is monotonically decreasing just in case

$$\forall s_1.\, \forall s_2.\, (\forall x.\, s_2(x) \Rightarrow s_1(x)) \Rightarrow q(s_1) \Rightarrow q(s_2). \tag{4.70}$$

Here s_1 and s_2 are properties, that is, of type $np \to s$. According to this approximate generalization, the sentence (4.69a) has a linear-scope reading because the quantifier nobody is monotonically decreasing: letting q be $\lambda c.\, \forall x.\, \neg c(x)$ in (4.70) gives

$$\forall s_1.\, \forall s_2.\, (\forall x.\, s_2(x) \Rightarrow s_1(x)) \Rightarrow (\forall x.\, \neg s_1(x)) \Rightarrow (\forall x.\, \neg s_2(x)), \tag{4.71}$$

which is a valid formula in second-order predicate logic. For example, let s_1 be the property of seeing someone and s_2 be the property of seeing a student: because seeing a student entails seeing someone, if nobody has the property of seeing someone then nobody has the property of seeing a student.

By contrast, the sentence (4.69a) has no inverse-scope reading, and (4.69b) and (4.69c) have no reading whatsoever, because those contexts for anyone are not downward-entailing. For instance, the quantifier everyone is not monotonically decreasing: letting q be $\lambda c.\, \forall x.\, c(x)$ in (4.70) gives

$$\forall s_1.\, \forall s_2.\, (\forall x.\, s_2(x) \Rightarrow s_1(x)) \Rightarrow (\forall x.\, s_1(x)) \Rightarrow (\forall x.\, s_2(x)), \tag{4.72}$$

which is not a valid formula in second-order predicate logic.

So far, we have presented the view that anyone is an existential quantifier that can only be used in certain contexts. In fact, it has been controversial in linguistics for decades whether anyone quantifies existentially, universally, both, or neither. Below we avoid that semantic debate and concentrate on the syntactic relationship between nobody and anyone in (4.69a). We say that nobody *licenses* anyone: unlike everyone in (4.69b) and Alice in (4.69c), nobody in (4.69a) creates a context in which anyone can occur.

Perhaps because it spans syntax and semantics, polarity sensitivity has been a popular linguistic phenomenon to analyze in logically-motivated approaches to grammar, such as the type-logical (Bernardi 2002; Bernardi and Moot 2001), categorial (Dowty 1994), and lexical-functional (Fry 1997, 1999) traditions. Fry (1997, 1999), Bernardi (2002), and Bernardi and Moot (2001) all split the clause type s into several types, some of which entail others. Fry (1997, 1999) distinguishes between the types s and $\ell \multimap (s \otimes \ell)$ in linear logic, the first of which entails the second. Analogously, Bernardi (2002) and Bernardi and Moot (2001) distinguish between different unary modes applied to s in multimodal type-logical grammar.

A simplistic version of Bernardi's analysis is to distinguish between the clause types s ("neutral clause") and $\Box_p^{\downarrow}\Diamond_p s$ ("negative clause"). Here $\Diamond_p$ is a new unary mode (the letter p stands for "polarity"), and we abbreviate $\Box_p^{\downarrow}\Diamond_p s$ to s^-. We choose the type $\Box_p^{\downarrow}\Diamond_p s$ so that $s \vdash s^-$ is a theorem in multimodal type-logical grammar.

$$\dfrac{\dfrac{\dfrac{}{s \vdash s}\,\mathrm{Id}}{\langle s \rangle_p \vdash \Diamond_p s}\,\Diamond_p \mathrm{I}}{s \vdash \Box_p^{\downarrow}\Diamond_p s}\,\Box_p^{\downarrow}\mathrm{I} \tag{4.73}$$

Splitting the clause type like this helps us account for polarity sensitivity, because we can now assign different types to different quantifiers in the lexicon.

$$\textsf{everyone} \vdash s /\!\!/ (np \backslash\!\!\backslash s) \tag{4.74}$$

$$\textsf{nobody} \vdash s /\!\!/ (np \backslash\!\!\backslash s^-) \tag{4.75}$$

$$\textsf{anyone} \vdash s^- /\!\!/ (np \backslash\!\!\backslash s^-) \tag{4.76}$$

The type of everyone is unchanged: it takes scope over a neutral clause to form a neutral clause. The types of nobody and anyone involve the newly introduced s^-: they both take scope over a negative clause, but nobody forms a neutral clause whereas anyone forms a negative clause. As the proof (4.73) shows, a neutral clause can be converted to a negative clause. Thus, as Figure 4.15 shows, anyone can take scope in $np, (\textsf{saw}, \textsf{anyone})$ to form a negative clause s^-, even though the verb saw produces a neutral clause s initially. As before, we consider a derivation complete if it culminates in the type s, not s^-. The fact that $np, (\textsf{saw}, \textsf{anyone})$

$$\dfrac{\mathsf{anyone} \vdash s^- /\!\!/ (np \backslash\!\!\backslash s^-) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots \text{ (4.14) on page 99}}{np, (\mathsf{saw}, np) \vdash s}\Diamond_p \mathrm{I}}{\langle np, (\mathsf{saw}, np)\rangle_p \vdash \Diamond_p s}\Box_p^{\downarrow} \mathrm{I}}{np, (\mathsf{saw}, np) \vdash s^-}\,\mathrm{Root}}{(np, (\mathsf{saw}, np)) \circledcirc 1 \vdash s^-}\,\mathrm{Right}'}{(\mathsf{saw}, np) \circledcirc (1, np) \vdash s^-}\,\mathrm{Right}'}{np \circledcirc ((1, np), \mathsf{saw}) \vdash s^-}\backslash\!\!\backslash \mathrm{I} \qquad (1, np), \mathsf{saw} \vdash np \backslash\!\!\backslash s^-}{\mathsf{anyone} \circledcirc ((1, np), \mathsf{saw}) \vdash s^-}/\!\!/ \mathrm{E}$$

$$\dfrac{\dfrac{\dfrac{\mathsf{anyone} \circledcirc ((1, np), \mathsf{saw}) \vdash s^-}{(\mathsf{saw}, \mathsf{anyone}) \circledcirc (1, np) \vdash s^-}\,\mathrm{Right}'}{(np, (\mathsf{saw}, \mathsf{anyone})) \circledcirc 1 \vdash s^-}\,\mathrm{Right}'}{np, (\mathsf{saw}, \mathsf{anyone}) \vdash s^-}\,\mathrm{Root}$$

Figure 4.15. The negative clause (incomplete sentence) *np* saw anyone

only has the type s^- (as derived above) and not s predicts that a clause like *Alice saw anyone (4.69c) is not a grammatical sentence by itself. Moreover, because there is no way to convert a negative clause to a neutral clause, everyone cannot take scope over a negative clause, so *Everyone saw anyone (4.69b) is ruled out as well. However, nobody (unlike Alice or everyone) can take scope over anyone to form a complete (neutral) clause. Thus even our simplistic rendering of Bernardi's account predicts correctly that Nobody saw anyone (4.69a) has a linear-scope reading (but no inverse-scope reading). We also predict correctly that the sentence

(4.77) Nobody introduced everyone to anyone.

has no linear-scope reading, and that it does have an interpretation on which nobody scopes over anyone, then everyone. (Kroch (1974), Linebarger (1980), and Szabolcsi (2004) discuss such *intervention* cases.)

This picture is complicated by the empirical observation that the syntactic relationship between the licensor and the licensee is restricted. For example, (4.69a) above is acceptable—nobody manages to license anyone—but

(4.78) *Anyone saw nobody.

is not. As the contrasts below further illustrate, the licensor usually must precede the licensee. (The examples in (4.80) show that c-command is not at issue, because neither nobody nor anyone c-commands the other. The examples in (4.81) show

that subject-object asymmetry is not the culprit either, because the only subject is I.)

(4.79) a. Alice didn't visit anyone.
b. *Anyone didn't visit Alice.

(4.80) a. Nobody's mother saw anyone's father.
b. *Anyone's mother saw nobody's father.

(4.81) a. I gave nothing to anyone.
b. *I gave anything to nobody.
c. I gave nobody anything.
d. *I gave anyone nothing.

These and other examples show that the syntactic relationships allowed between licensor and licensee for polarity are similar to those allowed between antecedent and pronoun for anaphora.

Because Fry, Bernardi, and Bernardi and Moot focus on quantification and scope, they easily characterize how nobody must take scope over anyone, but they leave it a mystery why nobody must precede anyone. In particular, their accounts wrongly accept all of (4.78)–(4.81). Ladusaw (1979) notes this mystery in his Inherent Scope Condition: "If the negative polarity item is clausemate with the trigger, the trigger must precede" (Section 4.4). Likewise, Fry (1999; Section 8.2) faults current accounts of polarity sensitivity for ignoring linear order. Ladusaw goes on to speculate that this left-right requirement is related to quantifier scope and sentence processing:

> I do not at this point see how to make this part of the Inherent Scope Condition follow from any other semantic principle. This may be because the left-right restriction, like the left-right rule for unmarked scope relations, should be made to follow from the syntactic and semantic processing of sentences .... (Section 9.2)

This speculation is our claim: the preference for linear order in quantification and the requirement that a polarity licensor precede its licensee are both due to left-to-right evaluation order. Our type-logical treatment of in-situ quantification and its application to multiple linguistic side effects provide precisely the missing link between linear order and polarity sensitivity.

Recall from Section 4.4 that inverse scope can be generated, even under a regime of left-to-right evaluation, using the Unquote rule in (4.47) on page 112. The crucial step, as illustrated in Figure 4.11 on page 113, is to unquote the clause produced by the narrower-scope quantifier. In that example, everyone can take inverse scope over someone, because someone produces a neutral clause s. A neutral clause can be unquoted because its type s is shorthand for

$\Diamond_u s'$, which is enclosed in $\Diamond_u$. By contrast, a negative clause cannot be unquoted because its type s^- is shorthand for $\Box_p^{\downarrow}\Diamond_p\Diamond_u s'$, which is not enclosed in $\Diamond_u$. Given that anyone produces s^-, then, inverse scope over anyone is impossible. This prediction correctly rules out the unacceptable examples in (4.78)–(4.81) while leaving (4.77) available.

This explanation for linear order in polarity sensitivity is further developed elsewhere (Shan 2004b), including treating someone as a *positive polarity item*. We are not aware of any other account of polarity sensitivity that unifies its syntactic properties with crossover as we do using left-to-right evaluation.

4.5.4. Reversing evaluation order. One way to check that our implementation of evaluation order unifies linear-order effects in crossover, superiority, and polarity is to impose right-to-left evaluation and see what happens. Because our Left and Right′ rules embody left-to-right evaluation order, we can reverse default evaluation order while leaving all other aspects of the system unchanged.

Specifically, suppose that we replace the Left and Right′ rules in (4.44) on page 110 with

$$\text{(4.82)}\qquad \frac{\Gamma[\Theta \circledcirc (\langle\Pi\rangle, \Delta)] \vdash T}{\Gamma[(\Theta, \langle\Pi\rangle) \circledcirc \Delta] \vdash T}\ \text{Left}' \qquad \frac{\Gamma[(\Theta, \Pi) \circledcirc \Delta] \vdash T}{\Gamma[\Pi \circledcirc (\Delta, \Theta)] \vdash T}\ \text{Right},$$

and the Trace Left and Trace Right′ rules in (4.60) and (4.61) on page 120 with[13]

$$\text{(4.83)}\qquad \frac{\Gamma[\mathrm{A}_{,?}\,(\Theta \circledcirc (\langle\Pi\rangle, \Delta))] \vdash T}{\Gamma[\mathrm{A}_{,?}\,(\Theta, (\langle\Pi\rangle \circledcirc \Delta))] \vdash T}\ \text{Trace Left}'$$

$$\text{(4.84)}\qquad \frac{\Gamma[\mathrm{A}_{,?}\,(\Pi \circledcirc (\Delta, \Theta))] \vdash T}{\Gamma[\mathrm{A}_{,?}\,(\Pi, (\Theta \circledcirc \Delta))] \vdash T}\ \text{Trace Right}.$$

We can still derive linear and inverse scope for quantifiers, but the predictions concerning the crossover examples in (4.48) on page 114, the superiority examples in (4.55) on page 119, and the polarity examples in (4.78)–(4.81) on pages 126–127 reverse: an antecedent must follow any pronoun it binds, a wh-trace must follow any in-situ wh-phrase, and a polarity licensor must follow any polarity item it licenses.

[13] As one would expect from Footnote 12.

CHAPTER 5

Delimited duality in programming languages

In this chapter, we apply the study of in-situ quantification in natural languages to the study of delimited control in programming languages. We present a programming language with delimited control and equip it with a duality, that is, an involution on programs and their execution. This duality is of interest for two reasons.

First, the duality exchanges call-by-value and call-by-name, two evaluation orders long studied in programming languages. Thus, not only can call-by-value and call-by-name be encoded in each other (Plotkin 1975), but the same encoding works in both directions in a suitable language, so a programmer can choose more easily between them. We are not the first to note that call-by-value and call-by-name are dual to each other (Filinski 1989a,b; Danos et al. 1995; Curien and Herbelin 2000; Selinger 2001; Wadler 2003). Rather, we are the first to construct such a duality in the presence of *delimited* rather than *undelimited* control. This difference arises because we draw our inspiration from quantification in natural language: the context over which a quantifier takes scope is delimited.

Second, the duality is the first to exchange contexts using *let* (defined in (2.32) on page 34) with expressions using *shift* (defined in Figure 3.9 on page 76). Although both features are useful, let is much more familiar than shift to the typical programmer, who can thus understand shift by way of let and use shift more effectively. The intuition underlying this duality between let and shift is that a shift-expression is a function that applies outward and a let-context is a function that applies inward, like a quantifier and its scope in natural language.

Two ideas from type-logical grammar motivate the design of our programming language. The first idea is that, given a decomposition of an expression into a subexpression and a context, we can treat both the subexpression and the context as binary trees and depict them as unitrivalent graphs. In particular, according to the structural rules proposed in Section 4.1 in type-logical grammar, the same default binary mode builds up subexpressions in an utterance as well as the delimited contexts over which they may take scope.

The second idea is that, when two syntactic structures combine, either constituent may be a function that takes the other as argument. In particular, the / E and \ E rules in Section 2.4 both use the default mode to combine the premises' type environments, so the direction of function application—that is, which con-

stituent applies to which—can be ambiguous given only the conclusion's type environment.

For the default mode, which prepends one constituent to another, this bidirectional function application yields a duality between left and right. That is, the reversal involution $-$ on types and type environments defined by

$$\begin{aligned} -A &= A, \\ -(T_2/T_1) &= -T_1\backslash -T_2, & -(x:T) &= x:-T, \\ -(T_1\backslash T_2) &= -T_2/-T_1, & -(\Gamma,\Delta) &= -\Delta,-\Gamma, \\ -(T_2/_m T_1) &= -T_2/_m -T_1, & -(\Gamma,_m \Delta) &= -\Gamma,_m -\Delta \\ -(T_1\backslash_m T_2) &= -T_1\backslash_m -T_2, \end{aligned} \tag{5.1}$$

(for any non-default mode m) extends trivially to an involution on grammars and their derivations, by flipping the direction of function application. For the $\circledcirc$ mode, which plugs one constituent into another, the same bidirectionality yields a duality between inside and outside: the context may apply to the subexpression, as well as vice versa. That is, the reversal involution $\neg$ on types and type environments defined by

$$\begin{aligned} \neg A &= A, \\ \neg(T_2 \mathbin{/\!\!/} T_1) &= \neg T_1 \mathbin{\backslash\!\!\backslash} \neg T_2, & \neg(x:T) &= x:\neg T, \\ \neg(T_1 \mathbin{\backslash\!\!\backslash} T_2) &= \neg T_2 \mathbin{/\!\!/} \neg T_1, & \neg(\Gamma \circledcirc \Delta) &= \neg\Delta \circledcirc \neg\Gamma, \\ \neg(T_2/_m T_1) &= \neg T_2/_m \neg T_1, & \neg(\Gamma,_m \Delta) &= \neg\Gamma,_m \neg\Delta \\ \neg(T_1\backslash_m T_2) &= \neg T_1\backslash_m \neg T_2, \end{aligned} \tag{5.2}$$

(for any non-continuation mode m) again extends trivially to an involution on grammars and their derivations, by flipping the direction of function application. Our side-effect analogy enables us to transfer this involution to a programming language with delimited control.

5.1. Call-by-value versus call-by-name

As mentioned in Section 2.3.1, the simply-typed λ-calculus is confluent, even though the computation relation is not deterministic: the same expression can execute in a variety of ways to yield the same result. We can make execution deterministic by imposing an evaluation order, such as call-by-value, left-to-right evaluation as enforced in Section 3.1.2. As explained in Section 3.1.1, evaluation order affects the final outcome of programs with control operators like abort and shift. Furthermore, in the untyped λ-calculus, not all programs terminate as in the simply-typed λ-calculus, and evaluation order affects whether programs can or must terminate. For example, the untyped program

$$(\lambda y.\lambda z.\, z)((\lambda x.\, xx)(\lambda x.\, xx)) \tag{5.3}$$

Values $\Gamma \vdash_{\mathrm{V}} V$

$$\frac{\Gamma, x \vdash E}{\Gamma \vdash_{\mathrm{V}} \lambda x.\, E} \qquad \frac{}{\Gamma, x \vdash_{\mathrm{V}} x}$$

Expressions $\Gamma \vdash E$

$$\frac{\Gamma \vdash_{\mathrm{V}} V}{\Gamma \vdash V} \qquad \frac{\Gamma \vdash E}{\Gamma \vdash \#E} \qquad \frac{\Gamma, c_{\mathrm{D}} \vdash E}{\Gamma \vdash \xi c.\, E} \qquad \frac{\Gamma \vdash F \quad \Gamma \vdash E}{\Gamma \vdash FE} \qquad \frac{\Gamma, c_{\mathrm{D}} \vdash E}{\Gamma, c_{\mathrm{D}} \vdash \#(cE)}$$

Evaluation subcontexts $D[\,] : d$

$$\frac{}{[\,] : d} \qquad \frac{\vdash F \quad D[\,] : d}{D[F[\,]] : d} \qquad \frac{D[\,] : d \quad \vdash_{\mathrm{V}} V}{D[[\,]V] : d}$$

Evaluation contexts $C[\,]$

$$\frac{D[\,] : d}{D[\,]} \qquad \frac{C[\,] \quad D[\,] : d}{C[\#(D[\,])]}$$

Computation $\#E \rhd \#E'$

$$\begin{aligned} \#(C[(\lambda x.\, E')V]) &\rhd \#(C[E'\,\{x \mapsto V\}]) \\ \#(C[\#V]) &\rhd \#(C[V]) \\ \#(C[\#(D[\xi c.\, E'])]) &\rhd \#(C[\#E'\,\{c[\,] \mapsto D[\,]\}]) \end{aligned}$$

Figure 5.1. An untyped λ-calculus with shift and reset, enforcing call-by-value, argument-to-function evaluation

(based on (2.33) on page 35) computes to itself as well as the value

(5.4) $$\lambda z.\, z.$$

Hence it initiates an infinite sequence of computation steps as well as finite ones that conclude in a normal form. On one hand, if we restrict the computation relation to rule out the step from (5.3) to (5.4), for example by enforcing call-by-value evaluation, then we prevent the program from stopping. On the other hand, if we rule out the step from (5.3) to itself, then we force the program to stop right away.

In Sections 3.1 and 3.2, we introduced the delimited control constructs $\#E$ (called prompt or reset) and $\xi c.\, E$ (called shift) in a programming language based on the simply-typed λ-calculus. Figures 5.1 and 5.2 define two variants of the untyped λ-calculus with shift and reset, with the same set of expressions but different computation relations. Each of Figures 5.1 and 5.2 provides inference rules for four judgment forms. The judgment $\Gamma \vdash E$ classifies E as a well-formed

Values $\Gamma \vdash_{\mathrm{V}} V$

$$\frac{\Gamma, x \vdash E}{\Gamma \vdash_{\mathrm{V}} \lambda x.\, E}$$

Expressions $\Gamma \vdash E$

$$\frac{\Gamma \vdash_{\mathrm{V}} V}{\Gamma \vdash V} \qquad \frac{}{\Gamma, x \vdash x} \qquad \frac{\Gamma \vdash E}{\Gamma \vdash \#E} \qquad \frac{\Gamma, c_{\mathrm{D}} \vdash E}{\Gamma \vdash \xi c.\, E} \qquad \frac{\Gamma \vdash F \quad \Gamma \vdash E}{\Gamma \vdash FE} \qquad \frac{\Gamma, c_{\mathrm{D}} \vdash E}{\Gamma, c_{\mathrm{D}} \vdash \#(cE)}$$

Evaluation subcontexts $D[\,] : d$

$$\frac{}{[\,] : d} \qquad \frac{D[\,] : d \quad \vdash E}{D[[\,]E] : d}$$

Evaluation contexts $C[\,]$

$$\frac{D[\,] : d}{D[\,]} \qquad \frac{C[\,] \quad D[\,] : d}{C[\#(D[\,])]}$$

Computation $\#E \rhd \#E'$

$$\#(C[(\lambda x.\, E')E]) \rhd \#(C[E'\,\{x \mapsto E\}])$$
$$\#(C[\#V]) \rhd \#(C[V])$$
$$\#(C[\#(D[\xi c.\, E'])]) \rhd \#(C[\#E'\,\{c[\,] \mapsto D[\,]\}])$$

Figure 5.2. An untyped λ-calculus with shift and reset, enforcing call-by-name, context-to-function evaluation

expression in the type environment Γ. As a special case, the judgment $\Gamma \vdash_{\mathrm{V}} V$ classifies V as a value expression. A type environment Γ is a finite, possibly empty set of variables, some subscripted by $_{\mathrm{D}}$ (as in c_{D}) to indicate a context variable. The judgment $D[\,] : d$ classifies $D[\,]$ as an evaluation subcontext. The judgment $C[\,]$ classifies $C[\,]$ as an evaluation context: essentially, a sequence of subcontexts separated by control delimiters.

Compared to the programming language in Figure 3.9 on page 76, these variants are more suitable for our purposes in this chapter because they incorporate two changes. First, we consider an expression to be a complete program only if it is enclosed by #.[1] Thus we define the computation relation only for expressions

[1] If we were considering a delimited control operator (such as Gunter et al.'s *cupto* (1995, 1998)) that, unlike shift, allows removing a control delimiter around an expression that is not a value, then for an expression to be enclosed by a delimiter would no longer guarantee that the control operators in the expression execute successfully. It would then be no longer appropriate to consider only an expression enclosed by a delimiter as a complete program.

enclosed by #. Second, the two variants enforce two orders of evaluation. The variant in Figure 5.1 enforces *call-by-value* but *argument-to-function* evaluation. Argument-to-function evaluation means that, when a function F is applied to an argument E, evaluation proceeds from E to F. Because we write F to the left of E as usual, argument-to-function evaluation is right-to-left evaluation—the opposite of left-to-right evaluation as in Chapters 3 and 4. We consider the argument-to-function variant of call-by-value evaluation here because it turns out to be dual to standard ("context-to-function") *call-by-name* evaluation.[2] The latter evaluation order is shown in Figure 5.2. Although shift and reset have been proposed only for call-by-value languages in the literature, we think the system in Figure 5.2 is a reasonable proposal for shift and reset under call-by-name: it extends the call-by-name λ-calculus (Plotkin 1975) and otherwise tracks Figure 5.1.

As explained in Section 3.1.2, call-by-value evaluation means to substitute only values for variables and never evaluate a subexpression inside a body (that is, under a variable binding). By contrast, call-by-name evaluation means to substitute any expressions for variables but never evaluate a subexpression inside a body or an argument. For example, whereas call-by-value evaluation computes

$$\#((\lambda y.\, \lambda z.\, z)((\lambda x.\, xx)(\lambda x.\, xx))) \tag{5.5}$$

to itself only, call-by-name evaluation computes it to

$$\#(\lambda z.\, z) \tag{5.6}$$

only. To take another example, the program

$$\#(\mathsf{let}\ \xi c.\, 1\ \mathsf{be}\ x.\, 2), \tag{5.7}$$

which is shorthand for

$$\#((\lambda x.\, 2)(\xi c.\, 1)), \tag{5.8}$$

computes to #1 under call-by-value evaluation, but #2 under call-by-name evaluation.[3]

The analysis of in-situ quantification in Chapter 4 helps us understand the difference between call-by-value and call-by-name, by treating the plugging of an expression into a subcontext as a mode of syntactic combination (namely ◎) and by visualizing syntactic combination in terms of unitrivalent graphs. To take a simple example, the program (5.7) is the result of plugging the expression $\xi c.\, 1$

[2] Call-by-value, function-to-argument evaluation is dual to a nonstandard ("function-to-context") kind of call-by-name evaluation.

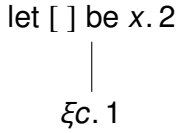
[3] Often "call-by-name" is taken to mean that one may substitute any expressions for variables, and may also evaluate a subexpression inside an argument or even a body. Call-by-name as defined here is a more restrictive—in fact deterministic—computation relation.

into the subcontext $\mathsf{let}\ [\ ]\ \mathsf{be}\ x.\,2$ inside a control delimiter. Using notation from Chapter 4, we can write the program roughly as

$$\#(\xi c.\,1 \circledcirc \mathsf{let}\ [\ ]\ \mathsf{be}\ x.\,2). \tag{5.9}$$

Intuitively, to the left of $\circledcirc$ is a structure with the syntactic type $\mathrm{int}/\!\!/T$: it combines with any type T to the right (that is, outside) to produce 1. To the right of $\circledcirc$ is a structure with the syntactic type $T\backslash\!\!\backslash\mathrm{int}$: it combines with any type T to the left (that is, inside) to produce 2. Because T can be any type for each of these constituents, they can combine by either the $/\!\!/$ E rule or the $\backslash\!\!\backslash$ E rule. Semantically speaking, the constant function returning 1 can apply to the constant function returning 2, as well as vice versa. The duality in (5.2) on page 130 makes the direction of function application ambiguous in (5.9), which we can visualize as the following trivial graph.

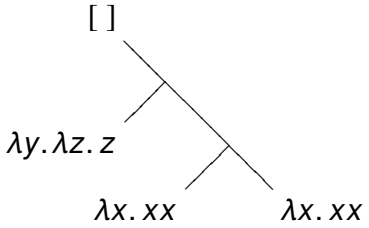
(5.10)



To break the symmetry, we can restrict the direction of function application, as follows. If we prohibit taking an argument whose type is of the form $T_2/\!\!/T_1$, then $\mathsf{let}\ [\ ]\ \mathsf{be}\ x.\,2$ can no longer apply to the non-value $\xi c.\,1$ by the $\backslash\!\!\backslash$ E rule, and (5.9) means 1 only, as call-by-value specifies. Dually, if we prohibit taking an argument whose type is of the form $T_1\backslash\!\!\backslash T_2$, then $\xi c.\,1$ can no longer apply to the non-subcontext $\mathsf{let}\ [\ ]\ \mathsf{be}\ x.\,2$ by the $/\!\!/$ E rule, and (5.9) means 2 only, as call-by-name specifies. Intuitively, a shift expression like $\xi c.\,1$ is a function that applies outward to the subcontext it is plugged into, as a quantificational noun phrase does. Dually, a let subcontext like $\mathsf{let}\ [\ ]\ \mathsf{be}\ x.\,2$ is a function that applies inward to the subexpression plugged into it, as a gapped clause does. Call-by-value means that the subexpression takes priority in applying to the subcontext, whereas call-by-name means that the subcontext takes priority in applying to the subexpression.

To visualize more complex programs, we need trivalent nodes in our graphs. For example, the program (5.5) on page 133 corresponds to the graph below.

(5.11)



A trivalent node indicates function application. Because the λ-calculus (unlike

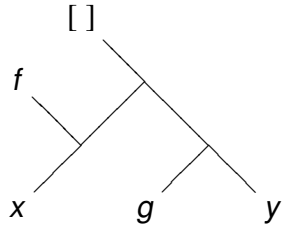
the Lambek calculus) distinguishes a function applied by putting it to the left of the argument, the trivalent nodes in this chapter (unlike those in Chapter 4) also distinguishes a function applied, by putting its edge perpendicular to the two other edges.

In general, a λ-expression corresponds to a "pipeline" that connects an argument at one end (such as $\lambda x.\, xx$ in the lower-right corner above) to a subcontext at the other end (such as [ ] in the upper-left corner above). Along the pipeline are zero or more functions (two above), which combine successively with the end argument to yield what plugs into the end subcontext, or dually, combine successively with the end subcontext to yield what the end argument plugs into. Each of these functions in turn forms one side of another pipeline. For example, a trivially short pipeline connects the function $\lambda y.\, \lambda z.\, z$ above to the subcontext $[\,](\lambda x.\, xx)(\lambda x.\, xx)$. Similarly, depicting the program

(5.12) $$\#((fx)(gy))$$

as the graph

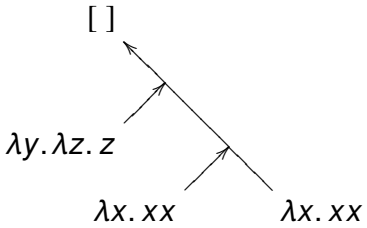
(5.13)



makes clear that a pipeline with just the function f connects x to the subcontext $[\,](gy)$.

One way to specify the order of evaluation in a graph like these is to orient the pipelines: for each pipeline, pick one end as the start and the other as the finish. For example, we could orient the graph (5.11) as

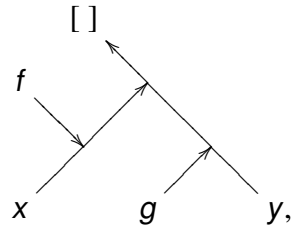
(5.14)



(in which each arrowhead orients one or more colinear edges, which form a pipeline), then consider the parts of the graph for evaluation from start to finish. That is, we first consider the argument $\lambda x.\, xx$ (which is already a value), then consider the function $\lambda x.\, xx$ (which applies). Because $(\lambda x.\, xx)(\lambda x.\, xx)$ β-reduces to itself, we never get to consider the function $\lambda y.\, \lambda z.\, z$ further along the pipeline.

Similarly, if we orient the graph (5.13) as

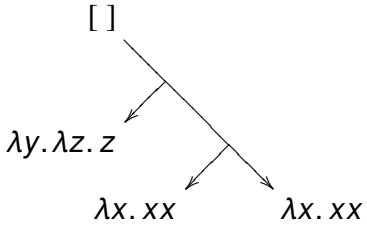
(5.15)



it means to apply g to y first, then apply f to x, and finally apply fx to gy. Roughly, to orient every pipeline towards the "root node" [ ] is to mandate call-by-value, argument-to-function evaluation.

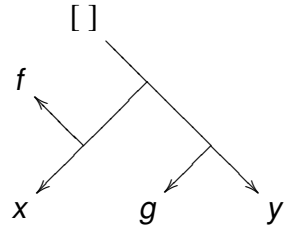
Dually, we can orient every pipeline away from the root [ ]. In the case of (5.11), it means to apply $\lambda y.\, \lambda z.\, z$ first, yielding the value $\lambda z.\, z$.

(5.16)



In the case of (5.13), it means to apply f first, then the function resulting from fx, without inspecting gy.

(5.17)



Roughly, to orient every pipeline away from the "root node" [ ] is to mandate call-by-name, context-to-function evaluation.

In the next section, we formalize these intuitions and make the duality between call-by-value and call-by-name explicit in an extended programming language with delimited control.

5.2. Delimited control in a dual calculus

Figure 5.3 defines the *dual calculus*, a programming language in which each complete program, a *statement* G, is a pipeline in which zero or more intermediate stages connect a left end to a right end. The intermediate stages and two ends of a pipeline are separated by semicolons ; in between. Accordingly, we write a

Leaf terms $\Gamma \vdash_{\mathrm{L}} E;$

$$\frac{\Gamma,\, x;,\, ; c \vdash G}{\Gamma \vdash_{\mathrm{L}} x.(G).c;} \qquad \frac{}{\Gamma,\, x; \vdash_{\mathrm{L}} x;} \qquad \frac{}{\Gamma \vdash_{\mathrm{L}} [];} \qquad \frac{\Gamma \vdash E; \quad \Gamma \vdash ; D}{\Gamma \vdash_{\mathrm{L}} (E; -; D);} \qquad \frac{\Gamma \vdash G}{\Gamma \vdash_{\mathrm{L}} \#G;} \qquad \frac{\Gamma,\, ; c \vdash G}{\Gamma \vdash_{\mathrm{L}} (G).c;}$$

Leaf coterms $\Gamma \vdash_{\mathrm{L}} ; D$

$$\frac{\Gamma,\, x;,\, ; c \vdash G}{\Gamma \vdash_{\mathrm{L}} ; x.(G).c} \qquad \frac{}{\Gamma,\, ; c \vdash_{\mathrm{L}} ; c} \qquad \frac{}{\Gamma \vdash_{\mathrm{L}} ; []} \qquad \frac{\Gamma \vdash E; \quad \Gamma \vdash ; D}{\Gamma \vdash_{\mathrm{L}} ; (E; -; D)} \qquad \frac{\Gamma \vdash G}{\Gamma \vdash_{\mathrm{L}} ; \#G} \qquad \frac{\Gamma,\, x; \vdash G}{\Gamma \vdash_{\mathrm{L}} ; x.(G)}$$

Terms $\Gamma \vdash E;$

$$\frac{\Gamma \vdash_{\mathrm{L}} E;}{\Gamma \vdash E;} \qquad \frac{\Gamma \vdash E; \quad \Gamma \vdash E';}{\Gamma \vdash E; (E';);} \qquad \frac{\Gamma \vdash E; \quad \Gamma \vdash ; D'}{\Gamma \vdash E; (; D');}$$

Coterms $\Gamma \vdash ; D$

$$\frac{\Gamma \vdash_{\mathrm{L}} ; D}{\Gamma \vdash ; D} \qquad \frac{\Gamma \vdash ; D' \quad \Gamma \vdash ; D}{\Gamma \vdash ; (; D'); D} \qquad \frac{\Gamma \vdash E'; \quad \Gamma \vdash ; D}{\Gamma \vdash ; (E';); D}$$

Statements $\Gamma \vdash G$

$$\frac{\Gamma \vdash E; \quad \Gamma \vdash_{\mathrm{L}} ; D}{\Gamma \vdash E; D} \qquad \text{(These two rules are equivalent.)} \qquad \frac{\Gamma \vdash_{\mathrm{L}} E; \quad \Gamma \vdash ; D}{\Gamma \vdash E; D}$$

Evaluation contexts $C[\ ]$

$$\frac{}{[\ ]} \qquad \frac{C[\ ] \quad \Gamma \vdash ; D}{C[\#[\ ]; D]} \qquad \frac{\Gamma \vdash E; \quad C[\ ]}{C[E; \#[\ ]]}$$

Computation $G_1 \rhd G_2$

$$\begin{array}{rl} C[E; (E';); D] \rhd C[E'; (E; -; D)] & C[\#(E; []); D] \rhd C[E; D] \\ C[E; (; D'); D] \rhd C[(E; -; D); D'] & C[E; \#([]; D)] \rhd C[E; D] \\ C[x.(G).c; (E; -; D)] \rhd C[G\{x \mapsto E\}\{c \mapsto D\}] & C[(G).c; D] \rhd C[G\{c \mapsto D\}] \\ C[(E; -; D); x.(G).c] \rhd C[G\{x \mapsto E\}\{c \mapsto D\}] & C[E; x.(G)] \rhd C[G\{x \mapsto E\}] \end{array}$$

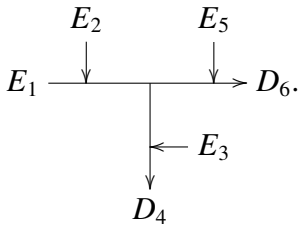
Figure 5.3. A dual calculus with delimited control operators

term E—an incomplete pipeline lacking a right end—with a dangling semicolon to the right, and a *coterm D*—an incomplete pipeline lacking a left end—with a dangling semicolon to the left. Each intermediate stage is in turn a term or a coterm, so a generic statement might look like

$$E_1; (E_2;); (; (E_3;); D_4); (E_5;); D_6 \tag{5.18}$$

and be depicted as a graph like

(5.19)



The statement (5.18) singles out the long horizontal pipeline in this graph as a distinguished trunk.

We use the metavariable x for a term variable, and the metavariable c for a coterm variable. This convention follows one of two possible, dual intuitions, namely that terms correspond to expressions in the λ-calculus, and coterms to subcontexts. A type environment Γ here is a finite set of variables, each marked a term variable $(x;)$ or a coterm variable $(;c)$. As defined by the judgment form $\Gamma \vdash_{\mathrm{L}} E;$, the possible left ends $E;$ are:

- an abstraction $x.(G).c;$,
- a variable $x;$,
- the unit $[];$,
- an activation $(E;-;D);$, related to *subtraction* (also known as *coimplication*) in logic (Crolard 2001, 2004),
- a delimited substatement $\#G;$,
- a shift-term $(G).c;$.

Dually, the judgment form $\Gamma \vdash_{\mathrm{L}} ;D$ defines the possible right ends $;D$ to be:

- an abstraction $;x.(G).c$,
- a variable $;c$,
- the unit $;[]$,
- an activation $;(E;-;D)$,
- a delimited substatement $;\#G$,
- a let-coterm $;x.(G)$.

An abstraction always takes two arguments x and c at once, one from each side of the pipeline. The abstraction can effectively take just one argument by reinstating the other right away, as in $x.(E;c).c$ and $x.(x;D).c$. This "dot" notation for abstraction is inspired by Wadler's (2003).

Be it as a term or a coterm, the unit [] represents the null subcontext [ ] in the λ-calculus. On the other hand, an activation $(E;-;D)$ represents a subcontext of the form $D[[\ ]E]$ in the λ-calculus. It can be thought of as a stack frame for invoking a function with two inputs $E;$ and $;D$, one of which is the argument and the other is the return address or continuation.

Translating expressions E to terms $\overline{E}$

$$\begin{aligned}
\overline{\lambda x.\, E} &= x.(\overline{E}; c).c; \\
\overline{x} &= x; \\
\overline{\#(cE)} &= \#(\overline{E}; c); \\
\overline{\xi c.\, E} &= (\overline{E}; []).c; \\
\overline{FE} &= \overline{E}; (\overline{F};); \\
\overline{\#E} &= \#(\overline{E}; []);
\end{aligned}$$

Translating evaluation subcontexts $D[\,]$ to coterms $\overline{D[\,]}$

$$\begin{aligned}
\overline{[\,]} &= ; [] \\
\overline{D[F[\,]]} &= ; (\overline{F};); \overline{D[\,]} \\
\overline{D[[\,]E]} &= ; (\overline{E}; -; \overline{D[\,]})
\end{aligned}$$

Figure 5.4. Translating the λ-calculus to the dual calculus

Graphically speaking, each left or right end that is not an activation corresponds to a leaf node in a graph. The leaf node corresponding to an abstraction, a delimited substatement, a shift-term, or a let-coterm contains a child graph within. An activation corresponds to coming to a stop at a T-intersection at the end of a pipeline. For example, the statement

$$(E_1; (E_2;); -; (E_5;); D_6); (E_3;); D_4 \tag{5.20}$$

corresponds to the same graph (5.19) as before, but with the long vertical pipeline singled out as the distinguished trunk.

The judgment form $\Gamma \vdash G$ defines statements G formally. A statement is simply the result of plugging a term E; and a coterm $; D$ together to form a complete pipeline $E; D$. Crucial to the notation and duality of our language is the fact that concatenating incomplete pipelines is an associative operation. Thus, for example, the statement (5.18) is the result of plugging together

- the term $E_1; (E_2;); (; (E_3;); D_4); (E_5;);$ and the coterm $; D_6$ (using the first rule for statements in Figure 5.3 on page 137),
- the term $E_1; (E_2;); (; (E_3;); D_4);$ and the coterm $; (E_5;); D_6$,
- the term $E_1; (E_2;);$ and the coterm $; (; (E_3;); D_4); (E_5;); D_6$, as well as
- the term $E_1;$ and the coterm $; (E_2;); (; (E_3;); D_4); (E_5;); D_6$ (using the second rule for statements in Figure 5.3 on page 137).

Figure 5.4 translates the λ-calculus with shift and reset to the dual calculus. More precisely, it translates expressions E to terms $\overline{E}$ and evaluation subcontexts $D[\,]$ to coterms $\overline{D[\,]}$. For example, the λ-expression (5.3) on page 130 translates to the term

$$\overline{(\lambda y.\,\lambda z.\,z)((\lambda x.\,xx)(\lambda x.\,xx))} = \begin{array}{l} x.(x;(x;)\,;c).c;\\ \quad (x.(x;(x;)\,;c).c;)\,;\\ \quad (y.(z.(z;c).c;c).c;)\,;. \end{array} \tag{5.21}$$

Enclosing the expression (5.3) in # to form the program (5.5) on page 133 in the λ-calculus corresponds to plugging together this term and the unit coterm $;[]$ in the dual calculus. The resulting statement can execute in the call-by-value way, yielding the following infinite loop of computation steps.

$$\begin{array}{ll} & x.(x;(x;)\,;c).c;\ \ (x.(x;(x;)\,;c).c;)\,;\ \ (y.(z.(z;c).c;c).c;)\,;\ \ [] \\ \rhd & x.(x;(x;)\,;c).c;\ \ (x.(x;(x;)\,;c).c;-;(y.(z.(z;c).c;c).c;)\,;[]) \\ \rhd & x.(x;(x;)\,;c).c;\ \ (x.(x;(x;)\,;c).c;)\,;\ \ (y.(z.(z;c).c;c).c;)\,;\ \ [] \end{array} \tag{5.22}$$

It can also execute in the call-by-name way, concluding in a normal form.

$$\begin{array}{ll} & x.(x;(x;)\,;c).c;\ \ (x.(x;(x;)\,;c).c;)\,;\ \ (y.(z.(z;c).c;c).c;)\,;\ \ [] \\ \rhd & y.(z.(z;c).c;c).c;\ \ (x.(x;(x;)\,;c).c;(x.(x;(x;)\,;c).c;)\,;-;[]) \\ \rhd & z.(z;c).c;\ \ [] \end{array} \tag{5.23}$$

We distinguish between call-by-value and call-by-name execution in the dual calculus only informally here. Section 5.3 below formalizes the distinction.

For another example, the λ-calculus program (5.7) on page 133 corresponds to the statement

$$(1;[]).c;\ \ (x.(2;c).c;)\,;\ \ [] \tag{5.24}$$

if we take 1 and 2 to translate to themselves. The call-by-value way to execute this statement binds c to the coterm $;(x.(2;c).c;)\,;[]$ and discards it.

$$(1;[]).c;\ \ (x.(2;c).c;)\,;\ \ [] \quad \rhd \quad 1;[] \tag{5.25}$$

The call-by-name way, on the other hand, binds x to the term $(1;[]).c;$ and discards it.

$$(1;[]).c;\ \ (x.(2;c).c;)\,;\ \ [] \quad \rhd \quad x.(2;c).c;\ \ ((1;[]).c;-;[]) \quad \rhd \quad 2;[] \tag{5.26}$$

Were we not to treat #(let $\xi c.\,1$ be $x.\,2$) as just shorthand for #$((\lambda x.\,2)(\xi c.\,1))$, we could translate the former expression more faithfully into the dual calculus as the statement

$$(1;[]).c;x.(2;[]), \tag{5.27}$$

$$\begin{aligned}
\neg(x.(G).c;) &= ;c.(\neg G).x & \neg(;x.(G).c) &= c.(\neg G).x;\\
\neg(x;) &= ;x & \neg(;c) &= c;\\
\neg([];) &= ;[] & \neg(;[]) &= [];\\
\neg((E;-;D);) &= ;(\neg D;-;\neg E) & \neg(;(E;-;D)) &= (\neg D;-;\neg E);\\
\neg(\#G;) &= ;\#\neg G & \neg(;\#G) &= \#\neg G;\\
\neg((G).c;) &= ;c.(\neg G) & \neg(;x.(G)) &= (\neg G).x;\\
\neg(E;(E';);) &= ;(;\neg E');\neg E & \neg(;(;D');D) &= \neg D;(\neg D';);\\
\neg(E;(;D');) &= ;(\neg D';);\neg E & \neg(;(E';);D) &= \neg D;(;\neg E');\\
\multicolumn{4}{c}{\neg(E;D) = \neg D;\neg E}
\end{aligned}$$

Figure 5.5. Duality in the dual calculus

formed by plugging together the shift-term $(1;[]).c$; and the let-coterm $;x.(2;[])$. These examples demonstrate that the computation relation for the dual calculus is not confluent.

The translations in Figure 5.4 on page 139 respect substitution: mutual induction on the structure of E' and $D'[\,]$ shows that

$$\overline{E'}\{x \mapsto \overline{E}\} = \overline{E'\{x \mapsto E\}}, \qquad \overline{E'}\{c \mapsto \overline{D[\,]}\} = \overline{E'\{c[\,] \mapsto D[\,]\}}, \tag{5.28}$$

$$\overline{D'[\,]}\{x \mapsto \overline{E}\} = \overline{D'[\,]\{x \mapsto E\}}, \quad \overline{D'[\,]}\{c \mapsto \overline{D[\,]}\} = \overline{D'[\,]\{c[\,] \mapsto D[\,]\}}. \tag{5.29}$$

The translation in Figure 5.4 is, as one would expect, only one of two dual translations possible. Figure 5.5 inductively defines an obvious involution $\neg$ between terms and coterms of the dual calculus, and among its statements. This involution respects the computation relation. Translating expressions E and subcontexts $D[\,]$ from the λ-calculus to coterms $\neg\overline{E}$ and terms $\neg\overline{D[\,]}$ in the dual calculus works just as well as translating them to terms $\overline{E}$ and coterms $\overline{D[\,]}$.

5.3. Restoring confluence

Figure 5.6 shows a restriction of the dual calculus that makes it confluent. It turns out that this restriction enforces call-by-value evaluation under the intuition (Figure 5.4 on page 139) that terms are subexpressions and coterms are subcontexts, but call-by-name evaluation under the dual (Figure 5.5 on page 141) intuition that terms are subcontexts and coterms are subexpressions. The judgment forms $\Gamma \vdash_{\mathrm{V}} V;$ and $\Gamma \vdash_{\mathrm{V}} ;U$ define certain terms and coterms, including abstraction, to be values V; and covalues $;U$. Furthermore, in an abstraction $(E;-;D)$, the term $E;$ must be a value, even though the coterm $;D$ need not be a covalue. The revised computation relation is crafted to only substitute a value

Values $\Gamma \vdash_{\mathrm{V}} V;$

$$\frac{\Gamma, x;, ; c \vdash G}{\Gamma \vdash_{\mathrm{V}} x.(G).c;} \qquad \frac{}{\Gamma, x; \vdash_{\mathrm{V}} x;} \qquad \frac{}{\Gamma \vdash_{\mathrm{V}} [];} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V; \quad \Gamma \vdash ; D}{\Gamma \vdash_{\mathrm{V}} (V; -; D);}$$

Covalues $\Gamma \vdash_{\mathrm{V}} ; U$

$$\frac{\Gamma, x;, ; c \vdash G}{\Gamma \vdash_{\mathrm{V}} ; x.(G).c}$$

Leaf terms $\Gamma \vdash_{\mathrm{L}} E;$

$$\frac{\Gamma \vdash_{\mathrm{V}} V;}{\Gamma \vdash_{\mathrm{L}} V;} \qquad \frac{\Gamma \vdash G}{\Gamma \vdash_{\mathrm{L}} \#G;} \qquad \frac{\Gamma, ; c \vdash G}{\Gamma \vdash_{\mathrm{L}} (G).c;}$$

Leaf coterms $\Gamma \vdash_{\mathrm{L}} ; D$

$$\frac{\Gamma \vdash_{\mathrm{V}} ; U}{\Gamma \vdash_{\mathrm{L}} ; U} \qquad \frac{}{\Gamma, ; c \vdash_{\mathrm{L}} ; c} \qquad \frac{}{\Gamma \vdash_{\mathrm{L}} ; []} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V; \quad \Gamma \vdash ; D}{\Gamma \vdash_{\mathrm{L}} ; (V; -; D)} \qquad \frac{\Gamma \vdash G}{\Gamma \vdash_{\mathrm{L}} ; \#G} \qquad \frac{\Gamma, x; \vdash G}{\Gamma \vdash_{\mathrm{L}} ; x.(G)}$$

Terms $\Gamma \vdash E;$

$$\frac{\Gamma \vdash_{\mathrm{L}} E;}{\Gamma \vdash E;} \qquad \frac{\Gamma \vdash E; \quad \Gamma \vdash E';}{\Gamma \vdash E; (E';);} \qquad \frac{\Gamma \vdash E; \quad \Gamma \vdash ; D'}{\Gamma \vdash E; (; D');}$$

Coterms $\Gamma \vdash ; D$

$$\frac{\Gamma \vdash_{\mathrm{L}} ; D}{\Gamma \vdash ; D} \qquad \frac{\Gamma \vdash ; D' \quad \Gamma \vdash ; D}{\Gamma \vdash ; (; D'); D} \qquad \frac{\Gamma \vdash E'; \quad \Gamma \vdash ; D}{\Gamma \vdash ; (E';); D}$$

Statements $\Gamma \vdash G$

$$\frac{\Gamma \vdash E; \quad \Gamma \vdash_{\mathrm{L}} ; D}{\Gamma \vdash E; D} \qquad \text{(These two rules are equivalent.)} \qquad \frac{\Gamma \vdash_{\mathrm{L}} E; \quad \Gamma \vdash ; D}{\Gamma \vdash E; D}$$

Evaluation contexts $C[\ ]$

$$\frac{}{[\ ]} \qquad \frac{C[\ ] \quad \Gamma \vdash ; D}{C[\#[\ ]; D]} \qquad \frac{\Gamma \vdash_{\mathrm{V}} V; \quad C[\ ]}{C[V; \#[\ ]]}$$

Computation $G_1 \rhd G_2$

$$C[V; (E';); D] \rhd C[E'; (V; -; D)] \qquad C[\#(V; []); D] \rhd C[V; D]$$
$$C[V; (; D'); D] \rhd C[(V; -; D); D'] \qquad C[V; \#([]; U)] \rhd C[V; U]$$
$$C[x.(G).c; (V; -; D)] \rhd C[G\{x \mapsto V\}\{c \mapsto D\}] \qquad C[(G).c; D] \rhd C[G\{c \mapsto D\}]$$
$$C[(V; -; D); x.(G).c] \rhd C[G\{x \mapsto V\}\{c \mapsto D\}] \qquad C[V; x.(G)] \rhd C[G\{x \mapsto V\}]$$

Figure 5.6. A dual calculus with delimited control operators, enforcing call-by-value evaluation

for a term variable, even though any coterm may still be substituted for a coterm variable. The notion of a covalue U only governs when to remove the control delimiter from a delimited substatement coterm. It is easy to check that this computation relation is deterministic, hence confluent.

Let us revisit the example statements above. This restriction on the dual calculus rules out the computation sequence (5.23) on page 140, leaving (5.22) on page 140. It also rules out the computation sequence (5.26) on page 140 in favor of (5.25) on page 140. The dual restriction (that is, conjugating Figure 5.6 by the involution $\neg$ in Figure 5.5 on page 141) makes the opposite choices among these computation sequences. Although the two dual restrictions of the dual calculus do not have the same expressions—an abstraction must have a value term under call-by-value, versus a covalue coterm under call-by-name—they both contain the image of the translation from the λ-calculus to the dual calculus in Figure 5.4 on page 139 (and the dual of that image).

To formalize how the dual calculus simulates the λ-calculus, let $C[\,]$ be a call-by-value evaluation context in the λ-calculus enclosed by #, say

$$C[\,] = \#(D_n[\ldots[\#(D_1[\#(D_0[\,])])]\ldots]). \tag{5.30}$$

We write $C^h[\,]$ (h for "head") for the subcontext $D_0[\,]$. We also write $C^t[\,]$ (t for "tail") for the context $\#(D_n[\ldots[\#(D_1[\,])]\ldots])$. Plugging a λ-expression E into $C[\,]$ gives the program $C[E]$. Correspondingly, we can plug the dual-calculus statement $\overline{E}; \overline{C^h[\,]}$ into the evaluation context $\overline{C^t[\,]}[\,]$ defined by

$$\overline{C^t[\,]}[\,] = \#(\ldots(\#[\,]; \overline{D_1[\,]})\ldots); \overline{D_n[\,]}. \tag{5.31}$$

(In particular, if $n = 0$ and $C^t[\,]$ is just $[\,]$, then $\overline{C^t[\,]}[\,]$ is also just $[\,]$.) We write $E@C[\,]$ for the resulting statement $\overline{C^t[\,]}[\overline{E}; \overline{C^h[\,]}]$.

As a special case, if V is a value expression in the call-by-value λ-calculus, then $\overline{V};$ is a value term in the call-by-value dual calculus, and the statement $\overline{V}; []$ is in normal form. As for computation, the following theorem maps each step in the λ-calculus to one or more steps in the dual calculus.

Theorem 5.1 (Call-by-value simulation). *Let $C_1[E_1] = C_2[E_2]$ in the call-by-value λ-calculus; in other words, suppose that $C_1[E_1]$ and $C_2[E_2]$ are two ways to decompose the same λ-expression. Suppose that $C_2[E_2] \rhd C'_2[E'_2]$ as follows, by one of the three kinds of computation steps in Figure 5.1 on page 131.*

E_2	E'_2	$C'_2[\,]$
$(\lambda x.\, E')V$	$E'\{x \mapsto V\}$	$C_2[\,]$
$\#V$	V	$C_2[\,]$
$\xi c.\, E'$	$E'\{c[\,] \mapsto C_2^h[\,]\}$	$C_2^t[\#[\,]]$

Then $E_1@C_1[\,] \rhd^+ E'_2@C'_2[\,]$ in the call-by-value dual calculus.

Proof. Inspecting the definition of values and evaluation contexts in Figure 5.1 on page 131 shows that E_1 and E_2, and accordingly $C_1[\ ]$ and $C_2[\ ]$, are related in one of three ways.

In the first case, $C[E_1] = E_2$ for some context $C[\ ]$; that is, $C_1[\ ] = C_2[C[\ ]]$ for some context $C[\ ]$. We consider a computation step of each kind in turn.

- If $E_2 = (\lambda x.\, E')V$, then $C[\ ]$ must be $[\ ]$, $[\ ]V$, or $(\lambda x.\, E')[\ ]$, so $C_1^t[\ ] = C_2^t[\ ]$. The computation sequence

$$\begin{aligned} E_2@C_2[\ ] = \overline{C_2^t[\ ]}[\overline{V}; (x.(\overline{E'}; c).c;); \overline{C_2^h[\ ]}] &= V@C_2[(\lambda x.\, E')[\ ]] \\ \triangleright\ \overline{C_2^t[\ ]}[x.(\overline{E'}; c).c; (\overline{V}; -; \overline{C_2^h[\ ]})] &= (\lambda x.\, E')@C_2[[\ ]V] \\ \triangleright\ \overline{C_2^t[\ ]}[\overline{E'}\,\{x \mapsto \overline{V}\}\,; \overline{C_2^h[\ ]}] &= E'\,\{x \mapsto V\}\,@C_2[\ ] \\ = E_2'@C_2'[\ ] \end{aligned} \tag{5.32}$$

  in the dual calculus covers all three cases for $C[\ ]$. The second-to-last equality uses the fact that

$$\overline{E'}\,\{x \mapsto \overline{V}\} = \overline{E'\,\{x \mapsto V\}}, \tag{5.33}$$

  from (5.28) on page 141.
- If $E_2 = \#V$, then $C[\ ]$ must be $[\ ]$ or $\#[\ ]$. In either case, we have

$$\begin{aligned} E_1@C_1[\ ] &= E_2@C_2[\ ] \\ &= \overline{C_2^t[\ ]}[\#(\overline{V}; []); \overline{C_2^h[\ ]}] \\ &\triangleright\ \overline{C_2^t[\ ]}[\overline{V}; \overline{C_2^h[\ ]}] \\ &= E_2'@C_2'[\ ]. \end{aligned} \tag{5.34}$$

- If $E_2 = \xi c.\, E'$, then $C[\ ]$ must be $[\ ]$, and we have

$$\begin{aligned} E_1@C_1[\ ] &= E_2@C_2[\ ] \\ &= \overline{C_2^t[\ ]}[(\overline{E'}; []).c; \overline{C_2^h[\ ]}] \\ &\triangleright\ \overline{C_2^t[\ ]}[\overline{E'}\,\{c \mapsto \overline{C_2^h[\ ]}\}\,; []] \\ &= E'\,\{c[\ ] \mapsto \overline{C_2^h[\ ]}\}\,@C_2^t[\#[\ ]] \\ &= E_2'@C_2'[\ ]. \end{aligned} \tag{5.35}$$

  The second-to-last equality uses the fact that

$$\overline{E'}\,\{c \mapsto \overline{C_2^h[\ ]}\} = \overline{E'\,\{c[\ ] \mapsto C_2^h[\ ]\}}, \tag{5.36}$$

  again from (5.28) on page 141.

In the second case, $E_1 = C[E_2]$ for some context $C[\ ]$; that is, $C_2[\ ] = C_1[C[\ ]]$ for some context $C[\ ]$. We proceed by induction on the structure of E_1 (with E_2

as the base case), in other words on the structure of $C[\ ]$ outside in (with $[\ ]$ as the base case).

- If E_1 is E_2 and $C[\ ]$ is $[\ ]$, then we have the first case already dealt with above.
- If E_1 is $F(C_0[E_2])$ or $\#(C_0[E_2])$, and $C[\ ]$ is accordingly $F(C_0[\ ])$ or $\#(C_0[\ ])$, then Figure 5.4 on page 139 confirms that $E_1@C_1[\ ]$ is the same statement as $C_0[E_2]@C_1[F[\ ]]$ or $C_0[E_2]@C_1[\#[\ ]]$. Hence $E_1@C_1[\ ] \rhd^+ E_2'@C_2'[\ ]$ by the induction hypothesis.
- If E_1 is $(C_0[E_2])V$ and $C[\ ]$ is $(C_0[\ ])V$, then

$$\begin{aligned} E_1@C_1[\ ] &= \overline{C_1^t[\ ]}[\overline{V}; (\overline{C_0[E_2]};); \overline{C_1^h[\ ]}] \\ &\rhd \overline{C_1^t[\ ]}[\overline{C_0[E_2]}; (\overline{V}; -; \overline{C_1^h[\ ]})] \\ &= C_0[E_2]@C_1[[\ ]V]. \end{aligned} \tag{5.37}$$

  By the induction hypothesis, $C_0[E_2]@C_1[[\ ]V] \rhd^+ E_2'@C_2'[\ ]$. Hence $E_1@C_1[\ ] \rhd^+ E_2'@C_2'[\ ]$ as well.

In the third, final case, E_1 is a value, and $C_2[\ ] = C_1'[C[\ ]E_1]$ and $C_1[\ ] = C_1'[C[E_2][\ ]]$ for some context $C[\ ]$. In this case, $E_1@C_1[\ ]$ is the same statement as $C[E_2]E_1@C_1'[\ ]$, and $C[E_2]E_1@C_1'[\ ] \rhd^+ E_2'@C_2'[\ ]$ by the first case already dealt with above. □

Corollary 5.2. *Whenever $C_1[E_1] \rhd E'$ in the call-by-value λ-calculus, there exists an expression E_2' and an evaluation context $C_2'[\ ]$ in the call-by-value λ-calculus, such that $E' = C_2'[E_2']$, and $E_1@C_1[\ ] \rhd^+ E_2'@C_2'[\ ]$ in the call-by-value dual calculus.*

Conversely, the following theorems show how each computation step in the dual calculus maps to either zero or one step in the λ-calculus.

Theorem 5.3. *If $E_1@C_1[\ ] \rhd G$ in the call-by-value dual calculus, then $G = E_2@C_2[\ ]$ for some expression E_2 and some evaluation context $C_2[\ ]$ in the call-by-value λ-calculus, such that either $C_1[E_1] = C_2[E_2]$ or $C_1[E_1] \rhd C_2[E_2]$. In any infinite sequence of steps $E_1@C_1[\ ] \rhd E_2@C_2[\ ] \rhd \cdots$, there is eventually an index i where $C_i[E_i] \rhd C_{i+1}[E_{i+1}]$.*

Proof. The first part of the theorem follows from inspecting the computation relation in Figure 5.6 on page 142. Only the first kind of steps listed there, namely

$$C[V; (E';); D] \rhd C[E'; (V; -; D)], \tag{5.38}$$

may occur without guaranteeing that $C_1[E_1] \rhd C_2[E_2]$. A step of this kind increases the number of minus signs by 1 while keeping the same number of semicolons in the statement. There is no infinite sequence of such steps because

every minus sign must appear between two semicolons, and there are only finitely many semicolons to start with. □

Theorem 5.4. *For any expression E and value V in the call-by-value λ-calculus, we have $\#E \rhd^* \#V$ if and only if $\overline{E};[] \rhd^* \overline{V};[]$. Moreover, $\#E$ begins an infinite sequence of computation steps if and only if $\overline{E};[]$ does.*

Proof. For any λ-expression E, by definition $\overline{E};[] = E@\#[\,]$.

[$\Rightarrow$] By induction via Corollary 5.2 on the number of computation steps in the call-by-value λ-calculus.

[$\Leftarrow$] By induction via Theorem 5.3 on the number of computation steps in the call-by-value dual calculus, and the fact that $E_2@C_2[\,] = \overline{V};[]$ implies $C_2[E_2] = \#V$. □

The dual of the call-by-value dual calculus, the call-by-name dual calculus, simulates the call-by-name λ-calculus with shift and reset. The analogous theorems below substantiate this simulation and our claim of duality. Their proofs follow the same structure as the call-by-value ones above, but are somewhat simpler because evaluation contexts are simpler in the call-by-name λ-calculus.

Theorem 5.5 (Call-by-name simulation). *Let $C_1[E_1] = C_2[E_2]$ in the call-by-name λ-calculus. Suppose that $C_2[E_2] \rhd C_2'[E_2']$ as follows, by one of the three kinds of computation steps in Figure 5.2 on page 132.*

E_2	E_2'	$C_2'[\,]$
$(\lambda x.\, E')E$	$E'\{x \mapsto E\}$	$C_2[\,]$
$\#V$	V	$C_2[\,]$
$\xi c.\, E'$	$E'\{c[\,] \mapsto C_2^h[\,]\}$	$C_2^t[\#[\,]]$

Then $E_1@C_1[\,] \rhd^+ E_2'@C_2'[\,]$ in the call-by-name dual calculus.

Corollary 5.6. *Whenever $C_1[E_1] \rhd E'$ in the call-by-name λ-calculus, there exists an expression E_2' and an evaluation context $C_2'[\,]$ in the call-by-name λ-calculus, such that $E' = C_2'[E_2']$, and $E_1@C_1[\,] \rhd^+ E_2'@C_2'[\,]$ in the call-by-name dual calculus.*

Theorem 5.7. *If $E_1@C_1[\,] \rhd G$ in the call-by-name dual calculus, then $G = E_2@C_2[\,]$ for some expression E_2 and some evaluation context $C_2[\,]$ in the call-by-name λ-calculus, such that either $C_1[E_1] = C_2[E_2]$ or $C_1[E_1] \rhd C_2[E_2]$. In any infinite sequence of steps $E_1@C_1[\,] \rhd E_2@C_2[\,] \rhd \cdots$, there is eventually an index i where $C_i[E_i] \rhd C_{i+1}[E_{i+1}]$.*

Theorem 5.8. *For any expression E and value V in the call-by-name λ-calculus, we have $\#E \rhd^* \#V$ if and only if $\overline{E};[] \rhd^* \overline{V};[]$. Moreover, $\#E$ begins an infinite sequence of computation steps if and only if $\overline{E};[]$ does.*

5.4. The continuation-passing-style transform

Figure 5.7 translates the call-by-value dual calculus into the plain λ-calculus without shift or reset. The translation consists of five sets of equations: from values V; to $\lfloor V;\rfloor$, from terms E; to $\lceil E;\rceil$, from covalues $;U$ to $\lfloor;U\rfloor$, from coterms $;D$ to $\lceil;D\rceil$, and from statements G to $[\![G]\!]$. A term or coterm translates to an abstraction or variable in the λ-calculus; a statement translates to an application. The term translation is essentially Fischer's continuation-passing-style transform for the call-by-value λ-calculus (1972); the coterm transform is essentially Plotkin's continuation-passing-style transform for the call-by-name λ-calculus (1975).

The statement translation $[\![G]\!]$ allows multiple translations for the same statement G because the same G can be decomposed into $E;D$ in $n+1$ different ways, if the pipeline G has n intermediate stages. Fortunately, these different translations all β-reduce to each other: it is easy to check that

$$\lceil E;(E';);\rceil\lceil;D\rceil \rhd \lceil E;\rceil\lceil;(E';);D\rceil, \quad \lceil E;(;D');\rceil\lceil;D\rceil \rhd \lceil E;\rceil\lceil;(;D');D\rceil. \tag{5.39}$$

Hence, induction on the structure of $E;$ gives $\lceil E;\rceil\lceil;D\rceil \rhd^* \lceil E';\rceil\lceil;D'\rceil$ whenever $E';$ is a leaf term and $E;D = E';D'$. We henceforth let $[\![E;D]\!]$ be $\lceil E';\rceil\lceil;D'\rceil$.

The translation of delimited substatements, shift-terms, and let-coterms introduces complex expressions $[\![G]\!]$ that behave differently under a call-by-value versus a call-by-name interpreter of the target language. Thus, strictly speaking, this translation produces not continuation-passing-style code but continuation-*composing*-style code. When the source program contains delimited substatements, shift-terms, or let-coterms, we intend for the output of this translation to be interpreted under call-by-value. To achieve indifference between call-by-value and call-by-name interpretation, we could simply apply another pass of the continuation-passing-style transform.

Theorem 5.9. *If $G \rhd G'$ in the call-by-value dual calculus, then $[\![G]\!] \rhd^+ [\![G']\!]$ in the call-by-value λ-calculus.*

Proof. By inspection of Figures 5.6 and 5.7. □

As one would expect, it is easy to check that, given any value $V;$ in the dual calculus, its continuation-passing-style transform $\lfloor V;\rfloor$ is a value in the λ-calculus, and $[\![V;[]]\!] \rhd^+ \lfloor V;\rfloor$.

Corollary 5.10. *If $G \rhd^* V;[]$ in the call-by-value dual calculus, then $[\![G]\!] \rhd^* \lfloor V;\rfloor$ in the call-by-value λ-calculus. If G begins an infinite sequence of computation steps, then $[\![G]\!]$ does too.*

Value translation $\lfloor V; \rfloor$

$$\begin{aligned}
\lfloor x.(G).c; \rfloor &= \lambda c.\, \lambda x.\, [\![G]\!] \\
\lfloor x; \rfloor &= x \\
\lfloor []; \rfloor &= \lambda f.\, f \\
\lfloor (V; -; D); \rfloor &= \lambda f.\, f \lceil ; D \rceil \lfloor V; \rfloor
\end{aligned}$$

Term translation $\lceil E; \rceil$

$$\begin{aligned}
\lceil V; \rceil &= \lambda c.\, c \lfloor V; \rfloor \\
\lceil \#G; \rceil &= \lambda c.\, c [\![G]\!] \\
\lceil (G).c; \rceil &= \lambda c.\, [\![G]\!] \\
\lceil E; (E';); \rceil &= \lambda c.\, \lceil E; \rceil (\lambda x.\, \lceil E'; \rceil (\lambda f.\, f c x)) \\
\lceil E; (; D'); \rceil &= \lambda c.\, \lceil E; \rceil (\lambda x.\, \lceil ; D' \rceil (\lambda f.\, f c x))
\end{aligned}$$

Covalue translation $\lfloor ; U \rfloor$

$$\lfloor ; x.(G).c \rfloor = \lambda c.\, \lambda x.\, [\![G]\!]$$

Coterm translation $\lceil ; D \rceil$

$$\begin{aligned}
\lceil ; U \rceil &= \lambda x.\, x \lfloor ; U \rfloor \\
\lceil ; c \rceil &= c \\
\lceil ; [] \rceil &= \lambda x.\, x \\
\lceil ; (V; -; D) \rceil &= \lambda f.\, f \lceil ; D \rceil \lfloor V; \rfloor \\
\lceil ; \#G \rceil &= \lambda x.\, [\![G]\!] x \\
\lceil ; x.(G) \rceil &= \lambda x.\, [\![G]\!] \\
\lceil ; (; D'); D \rceil &= \lambda x.\, \lceil ; D' \rceil (\lambda f.\, f \lceil ; D \rceil x) \\
\lceil ; (E';); D \rceil &= \lambda x.\, \lceil E'; \rceil (\lambda f.\, f \lceil ; D \rceil x)
\end{aligned}$$

Statement translation $[\![G]\!]$

$$[\![E; D]\!] = \lceil E; \rceil \lceil ; D \rceil$$

Figure 5.7. A continuation-passing-style transform for the dual calculus

CHAPTER 6

Conclusion

We have identified a general analogy between side effects in programming languages and in natural languages (Section 1.3). In particular, delimited control subsumes other computational side effects as quantification subsumes other linguistic side effects (Section 3.4).

- On one hand, in the quest for a denotational semantics that is sound and compositional, computational and linguistic side effects both make it impossible for every expression to just denote its reference or evaluation result (Section 1.6). In particular, in the standard denotational semantics for delimited control and quantification, the meaning of an expression is a function from a continuation (Section 3.2) or scope (Section 3.3).
- On the other hand, in the quest for an operational semantics that models how a computer executes a program or how a human processes an utterance, computational and linguistic side effects both make observable the order in which parts of an expression are evaluated. In particular, evaluation order governs what contexts can be accessed by a delimited control operator (Section 3.1) or a quantificational phrase (Section 4.3).

For both kinds of languages, the relation between denotational and operational semantics is a mind-body problem of sorts: how does an expression manage to be both a static product that stands alone mathematically and a dynamic action that takes place physically (Wadler 1997; Trueswell and Tanenhaus 2005)?

We have put this analogy to work in both directions. Drawing on the general notion of computational side effects, our work on natural language extends dynamic semantics (Groenendijk and Stokhof 1991; Heim 1982; Kamp 1981) from whole sentences to parts of a sentence, and from anaphora and existential quantification to other linguistic side effects. Drawing on the continuation semantics for delimited control, our new implementation of quantification in type-logical grammar (Section 4.2) is graphically motivated and does not move nearby constituents apart or distant constituents together (Section 4.1). Drawing on the programming-language concepts of evaluation order and multistage programming, our grammar unifies the preference for linear scope in quantification (Section 4.4) with the prohibition against crossover in anaphora, the superiority constraint on wh-questions, and the effect of linear order on polarity sensitivity (Section 4.5):

- Our treatment of crossover uses left-to-right evaluation to deal with problems for previous accounts based purely on c-command or linear order. It correctly predicts a complex pattern of interaction between anaphora and raised-wh questions, without any stipulation on both for the first time.
- Our explanation for superiority unifies it with crossover yet need not encode multiple-wh questions as functional questions.
- Our account of polarity sensitivity is the first to explain the effect of linear order by appealing to processing.

Further connections remain to be drawn and strengthened:

- to other linguistic side effects, such as intensionality, focus, and presuppositions (Section 1.6);
- to psycholinguistic evidence on human sentence processing, in particular how raised-wh questions are processed (Section 4.5.2);
- to types for multistage programming, especially modal types (Section 4.4); and
- to the philosophical notion of *illocution*.

These connections all serve a broader investigation of operational semantics for natural language.

Drawing from the duality between inside and outside in our treatment of linguistic quantification, our programming language with delimited control in Chapter 5 is the first to make subexpressions and subcontexts dual to each other.

- The duality exchanges call-by-value and call-by-name, two long-studied evaluation orders that had previously been shown dual in the presence of undelimited but not delimited control.
- The duality also exchanges the familiar *let* construct and the less-familiar *shift* construct, so that a programmer can understand the latter in terms of the former.

It remains to turn our type-logical grammar for linguistic quantification into a type system for this yet-untyped programming language. Such a type system would answer the open question of how to interpret delimited control logically under the formulas-as-types correspondence. We suspect that the answer will be a noncommutative intuitionistic logic, like type-logical grammar.

Bibliography

Abbott, Michael, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2003. Derivatives of containers. In *TLCA 2003: Proceedings of the 6th international conference on typed lambda calculi and applications*, ed. Martin Hofmann, 16–30. Lecture Notes in Computer Science 2701, Berlin: Springer-Verlag.

Abramsky, Samson, and Guy McCusker. 1997. Game semantics. Presented at the 1997 Marktoberdorf Summer School. 26 Sep. 2000, `http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/mdorf97.ps.gz`.

Ajdukiewicz, Kasimir. 1935. Die syntaktische Konnexität. *Studia Philosophica* 1: 1–27. English translation: Ajdukiewicz, Kasimir. 1967. Syntactic connexion. In *Polish logic, 1920–1939*, ed. Storrs McCall, 207–231. New York: Oxford University Press.

Altmann, G. T. M., ed. 1990. *Cognitive models of speech processing: Psycholinguistic and computational perspectives*. Cambridge: MIT Press.

Aoun, Joseph, and Yen-hui Audrey Li. 1993. *Syntax of scope*. Cambridge: MIT Press.

Areces, Carlos, and Raffaella Bernardi. 2003. In situ binding: A hybrid approach. In *Proceedings of ICoS-4: 4th international workshop on inference in computational semantics*.

Bar-Hillel, Yehoshua. 1953. A quasi-arithmetical notation for syntactic description. *Language* 29(1):47–58.

Barendregt, Henk P. 1981. *The lambda calculus: Its syntax and semantics*. Amsterdam: Elsevier Science.

Barker, Chris. 2001. Continuations: In-situ quantification without storage or type-shifting. In *Proceedings from Semantics and Linguistic Theory XI*, ed. Rachel Hastings, Brendan Jackson, and Zsofia Zvolensky. Ithaca: Cornell University Press.

———. 2002. Continuations and the nature of quantification. *Natural Language Semantics* 10(3):211–242.

———. 2004. Continuations in natural language. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke, 1–11. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.

———. 2005. Direct compositionality on demand. In *Direct compositionality*, ed. Chris Barker and Pauline Jacobson. New York: Oxford University Press.

In press.

Barker, Chris, and Chung-chieh Shan. 2005. Types as graphs: Continuations in type logical grammar. *Journal of Logic, Language and Information*. In press.

Barwise, Jon, and Robin Cooper. 1981. Generalized quantifiers and natural language. *Linguistics and Philosophy* 4:159–219.

Belnap, Nuel D., Jr. 1982. Display logic. *Journal of Philosophical Logic* 11(4): 375–417.

van Benthem, Johan F. A. K. 1983. The semantics of variety in categorial grammar. Report 83-29, Department of Mathematics, Simon Fraser University, Burnaby, BC.

———. 1988. The semantics of variety in categorial grammar. In *Categorial grammar*, ed. Wojciech Buszkowski, Witold Marciszewski, and Johan F. A. K. van Benthem, 37–55. Amsterdam: John Benjamins.

Bernardi, Raffaella. 2002. Reasoning with polarity in categorial type logic. Ph.D. thesis, Utrecht Institute of Linguistics (OTS), Utrecht University.

———. 2003. CTL: In situ binding. `http://www.let.uu.nl/~Raffaella.Bernardi/personal/q_solutions.pdf`.

Bernardi, Raffaella, and Richard Moot. 2001. Generalized quantifiers in declarative and interrogative sentences. *Journal of Language and Computation* 1(3): 1–19.

Biernacki, Dariusz, Olivier Danvy, and Kevin Millikin. 2005a. A dynamic continuation-passing style for dynamic delimited continuations. Report RS-05-16, BRICS, Denmark.

Biernacki, Dariusz, Olivier Danvy, and Chung-chieh Shan. 2005b. On the dynamic extent of delimited continuations. Report RS-05-13, BRICS, Denmark.

———. 2005c. On the dynamic extent of delimited continuations. *Information Processing Letters* 96(1):7–17.

Büring, Daniel. 2001. A situation semantics for binding out of DP. In *Proceedings from Semantics and Linguistic Theory XI*, ed. Rachel Hastings, Brendan Jackson, and Zsofia Zvolensky, 56–75. Ithaca: Cornell University Press.

———. 2004. Crossover situations. *Natural Language Semantics* 12(1):23–62.

Buszkowski, Wojciech. 1986. Generative capacity of nonassociative Lambek calculus. *Bulletin of the Polish Academy of Sciences: Mathematics* 34(7–8): 507–516.

Calcagno, Cristiano, Eugenio Moggi, and Tim Sheard. 2003. Closed types for a safe imperative MetaML. *Journal of Functional Programming* 13(3):545–571.

Carpenter, Bob. 1994. Quantification and scoping: A deductive account. In *Proceedings of the 13th West Coast Conference on Formal Linguistics*, ed. Raœl Aranovich, William Byrne, Susanne Preuss, and Martha Senturia. Stanford, CA: Center for the Study of Language and Information.

———. 1997. *Type-logical semantics*. Cambridge: MIT Press.

Chierchia, Gennaro. 1991. Functional WH and weak crossover. In *Proceedings of the 10th West Coast Conference on Formal Linguistics*, ed. Dawn Bates, 75–90. Stanford, CA: Center for the Study of Language and Information.

———. 1993. Questions with quantifiers. *Natural Language Semantics* 1(2): 181–234.

Chomsky, Noam. 1973. Conditions on transformations. In *A festschrift for Morris Halle*, ed. Stephen Anderson and Paul Kiparsky, 232–286. New York: Holt, Rinehart and Winston.

Church, Alonzo. 1932. A set of postulates for the foundation of logic. *Annals of Mathematics* II.33(2):346–366.

———. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5(2):56–68.

Cooper, Robin. 1983. *Quantification and syntactic theory*. Dordrecht: Reidel.

Crolard, Tristan. 2001. Subtractive logic. *Theoretical Computer Science* 254(1–2): 151–185.

———. 2004. A formulae-as-types interpretation of Subtractive Logic. *Journal of Logic and Computation* 14(4):529–570.

Curien, Pierre-Louis, and Hugo Herbelin. 2000. The duality of computation. In *ICFP '00: Proceedings of the ACM international conference on functional programming*, vol. 35(9) of *ACM SIGPLAN Notices*, 233–243. New York: ACM Press.

Dalrymple, Mary, John Lamping, Fernando Pereira, and Vijay A. Saraswat. 1999. Quantification, anaphora, and intensionality. In *Semantics and syntax in Lexical Functional Grammar: The resource logic approach*, ed. Mary Dalrymple, chap. 2, 39–89. Cambridge: MIT Press.

Danos, Vincent, Jean-Baptiste Joinet, and Harold Schellinx. 1995. LKQ and LKT: Sequent calculi for second order logic based upon dual linear decompositions of classical implication. In *Advances in linear logic*, ed. Jean-Yves Girard, Yves Lafont, and Laurent Regnier, 211–224. London Mathematical Society Lecture Note 222, Cambridge: Cambridge University Press.

Danvy, Olivier. 2004. On evaluation contexts, continuations, and the rest of the computation. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.

Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark. `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`.

———. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.

———. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

Danvy, Olivier, and Lasse R. Nielsen. 2001a. Defunctionalization at work. In *Proceedings of the 3rd international conference on principles and practice of declarative programming*, 162–174. New York: ACM Press.

———. 2001b. Defunctionalization at work. Report RS-01-23, BRICS, Denmark.

Davies, Rowan. 1996. A temporal logic approach to binding-time analysis. In *LICS '96: Proceedings of the 11th symposium on logic in computer science*, 184–195. Washington, DC: IEEE Computer Society Press.

Davies, Rowan, and Frank Pfenning. 1996. A modal analysis of staged computation. In *POPL '96: Conference record of the annual ACM symposium on principles of programming languages*, 258–270. New York: ACM Press.

———. 2001. A modal analysis of staged computation. *Journal of the ACM* 48(3):555–604.

Dayal, Veneeta. 1996. *Locality in wh quantification: Questions and relative clauses in Hindi*. Dordrecht: Kluwer.

Dowty, David R. 1992. 'Variable-free' syntax, variable-binding syntax, the natural deduction Lambek calculus, and the crossover constraint. In *Proceedings of the 11th West Coast Conference on Formal Linguistics*, ed. Jonathan Mead. Stanford, CA: Center for the Study of Language and Information.

———. 1994. The role of negative polarity and concord marking in natural language reasoning. In *Proceedings from Semantics and Linguistic Theory IV*, ed. Mandy Harvey and Lynn Santelmann. Ithaca: Cornell University Press.

Dunn, J. Michael. 1991. Gaggle theory: An abstraction of Galois connections and residuation with applications to negation and various logical operations. In *Logics in AI: European workshop JELIA '90*, ed. Jan van Eijck. Lecture Notes in Artificial Intelligence 478, Berlin: Springer-Verlag.

Dybvig, R. Kent, Simon L. Peyton Jones, and Amr Sabry. 2005. A monadic framework for delimited continuations. Tech. Rep. 615, Computer Science Department, Indiana University.

van Eijck, Jan. 1998. Programming with dynamic predicate logic. Report INS-R9810, Centrum voor Wiskunde en Informatica, Amsterdam. Also as Research Report CT-1998-06, Institute for Logic, Language and Computation, Universiteit van Amsterdam.

van Eijck, Jan, and Nissim Francez. 1995. Verb-phrase ellipsis in dynamic semantics. In *Applied logic: How, what, and why: Logical approaches to natural language*, ed. László Pólos and Michael Masuch. Dordrecht: Kluwer.

Felleisen, Matthias. 1987. The calculi of λ_v-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Computer Science Department, Indiana University. Also as Tech. Rep. 226.

———. 1988. The theory and practice of first-class prompts. In *POPL '88: Conference record of the annual ACM symposium on principles of programming*

languages, 180–190. New York: ACM Press.

Felleisen, Matthias, Daniel P. Friedman, Bruce F. Duba, and John Merrill. 1987. Beyond continuations. Tech. Rep. 216, Computer Science Department, Indiana University.

Felleisen, Matthias, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. 1988. Abstract continuations: A mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM conference on Lisp and functional programming*, 52–62. New York: ACM Press.

Filinski, Andrzej. 1989a. Declarative continuations: An investigation of duality in programming language semantics. In *Proceedings of category theory and computer science*, ed. David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné, 224–249. Lecture Notes in Computer Science 389, Berlin: Springer-Verlag.

———. 1989b. Declarative continuations and categorical duality. Master's thesis, DIKU, University of Copenhagen, Denmark. Also as Tech. Rep. 89/11.

———. 1994. Representing monads. In *POPL '94: Conference record of the annual ACM symposium on principles of programming languages*, 446–457. New York: ACM Press.

———. 1996. Controlling effects. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-96-119.

———. 1999. Representing layered monads. In *POPL '99: Conference record of the annual ACM symposium on principles of programming languages*, 175–188. New York: ACM Press.

Fischer, Michael J. 1972. Lambda-calculus schemata. In *Proceedings of ACM conference on proving assertions about programs*, vol. 7(1) of *ACM SIGPLAN Notices*, 104–109. New York: ACM Press. Also ACM SIGACT News 14.

Frege, Gottlob. 1891. Funktion und Begriff. Vortrag, gehalten in der Sitzung vom 9. Januar 1891 der Jenaischen Gesellschaft für Medizin und Naturwissenschaft, Jena: Hermann Pohle. English translation: Frege, Gottlob. 1980. Function and concept. In *Translations from the philosophical writings of Gottlob Frege*, ed. Peter Geach and Max Black, 3rd ed., 21–41. Oxford: Blackwell. Reprinted as: Frege, Gottlob. 1997. Function and concept. In *The Frege reader*, ed. Michael Beaney, 130–148. Oxford: Blackwell.

———. 1892. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik* 100:25–50. English translation: Frege, Gottlob. 1980. On sense and reference. In *Translations from the philosophical writings of Gottlob Frege*, ed. Peter Geach and Max Black, 3rd ed., 56–78. Oxford: Blackwell. Reprinted as: Frege, Gottlob. 1997. On Sinn and Bedeutung. In *The Frege reader*, ed. Michael Beaney, 151–171. Oxford: Blackwell.

Friedman, Daniel P., and David S. Wise. 1976. Cons should not evaluate its arguments. In *Automata, languages, and programming: 3rd international*

colloquium, ed. Sidney Michaelson and Robin Milner, 257–284. Edinburgh: Edinburgh University Press.

Fry, John. 1997. Negative polarity licensing at the syntax-semantics interface. In *Proceedings of the 35th annual meeting of the Association for Computational Linguistics and 8th conference of the European chapter of the Association for Computational Linguistics*, ed. Philip R. Cohen and Wolfgang Wahlster, 144–150. San Francisco: Morgan Kaufmann.

———. 1999. Proof nets and negative polarity licensing. In *Semantics and syntax in Lexical Functional Grammar: The resource logic approach*, ed. Mary Dalrymple, chap. 3, 91–116. Cambridge: MIT Press.

Gallier, Jean. 1991. Constructive logics, Part II: Linear logic and proof nets. Tech. Rep., University of Pennsylvania.

———. 1993. Constructive logics, Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science* 110:249–339.

Gapeyev, Vladimir, Michael Y. Levin, and Benjamin C. Pierce. 2002. Recursive subtyping revealed. *Journal of Functional Programming* 12(6):511–548.

Gardent, Claire. 1991. Dynamic semantics and VP-ellipsis. In *Logics in AI: European workshop JELIA '90*, ed. Jan van Eijck, 251–266. Lecture Notes in Artificial Intelligence 478, Berlin: Springer-Verlag.

Gibson, Edward, and Neal J. Pearlmutter. 1998. Constraints on sentence comprehension. *Trends in Cognitive Sciences* 2(7):262–268.

Girard, Jean-Yves. 1987. Linear logic. *Theoretical Computer Science* 50:1–101.

Girard, Jean-Yves, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Cambridge: Cambridge University Press.

Goubault-Larrecq, Jean. 1996a. On computational interpretations of the modal logic S4: I. Cut elimination. Interner Bericht 1996-35, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe.

———. 1996b. On computational interpretations of the modal logic S4: II. The $\lambda\mathrm{ev}Q$-calculus. Interner Bericht 1996-34, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe.

———. 1996c. On computational interpretations of the modal logic S4: IIIa. Termination, confluence, conservativity of $\lambda\mathrm{ev}Q$. Interner Bericht 1996-33, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe.

———. 1997. On computational interpretations of the modal logic S4: IIIb. Confluence, termination of the $\lambda\mathrm{ev}Q_H$-calculus. Rapport de Recherche 3164, INRIA.

Greiner, John. 1992. Programming with inductive and co-inductive types. Tech. Rep. CMU-CS-92-109, School of Computer Science, Carnegie Mellon University.

Groenendijk, Jeroen, and Martin Stokhof. 1984. Studies on the semantics of

questions and the pragmatics of answers. Ph.D. thesis, Universiteit van Amsterdam.

———. 1991. Dynamic predicate logic. *Linguistics and Philosophy* 14(1): 39–100.

———. 1997. Questions. In *Handbook of logic and language*, ed. Johan F. A. K. van Benthem and Alice G. B. ter Meulen, 1055–1124. Amsterdam: Elsevier Science.

de Groote, Philippe. 2001. Type raising, continuations, and classical logic. In *Proceedings of the 13th Amsterdam Colloquium*, ed. Robert van Rooy and Martin Stokhof, 97–101. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Gunter, Carl A., Didier Rémy, and Jon G. Riecke. 1995. A generalization of exceptions and control in ML-like languages. In *Functional programming languages and computer architecture: 7th conference*, ed. Simon L. Peyton Jones, 12–23. New York: ACM Press.

———. 1998. Return types for functional continuations. `http://pauillac.inria.fr/~remy/work/cupto/`.

Hamblin, Charles Leonard. 1973. Questions in Montague English. *Foundations of Language* 10:41–53.

Hardt, Daniel. 1999. Dynamic interpretation of verb phrase ellipsis. *Linguistics and Philosophy* 22(2):185–219.

Heim, Irene. 1982. The semantics of definite and indefinite noun phrases. Ph.D. thesis, Department of Linguistics, University of Massachusetts.

Henderson, Peter. 1982. Functional geometry. In *Proceedings of the 1982 ACM symposium on Lisp and functional programming*, 179–187. New York: ACM Press.

Hendriks, Herman. 1988. Type change in semantics: The scope of quantification and coordination. In *Categories, polymorphism and unification*, ed. Ewan Klein and Johan F. A. K. van Benthem, 96–119. Centre for Cognitive Science, University of Edinburgh.

Hepple, Mark. 1990. The grammar and processing of order and dependency: A categorial approach. Ph.D. thesis, Centre for Cognitive Science, University of Edinburgh.

———. 1992. Command and domain constraints in a categorial theory of binding. In *Proceedings of the 8th Amsterdam Colloquium*, ed. Paul Dekker and Martin Stokhof, 253–270. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Hieb, Robert, and R. Kent Dybvig. 1990. Continuations and concurrency. In *Proceedings of the 2nd ACM SIGPLAN symposium on principles and practice of parallel programming*, 128–136. New York: ACM Press.

Hindley, J. Roger, and Jonathan P. Seldin. 1986. *Introduction to combinators and*

λ-calculus. Cambridge: Cambridge University Press.

Hinze, Ralf, and Johan Jeuring. 2001. Weaving a web. *Journal of Functional Programming* 11(6):681–689.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12(10):576–580,583.

Hobbs, Jerry R., and Stanley J. Rosenschein. 1978. Making computational sense of Montague's intensional logic. *Artificial Intelligence* 9:287–306.

Hobbs, Jerry R., and Stuart M. Shieber. 1987. An algorithm for generating quantifier scopings. *Computational Linguistics* 13(1–2):47–63.

Hornstein, Norbert. 1995. *Logical form: From GB to Minimalism*. Oxford: Blackwell.

Howard, William A. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, ed. Jonathan P. Seldin and J. Roger Hindley, 479–490. San Diego, CA: Academic Press.

Huang, Cheng-Teh James. 1982. Logical relations in Chinese and the theory of grammar. Ph.D. thesis, Department of Linguistics and Philosophy, Massachusetts Institute of Technology.

Hudak, Paul, Tom Makucevich, Syam Gadde, and Bo Whong. 1996. Haskore music notation: An algebra of music. *Journal of Functional Programming* 6(3).

Huet, Gérard. 1997. The zipper. *Journal of Functional Programming* 7(5): 549–554.

Hung, Hing-Kai, and Jeffery I. Zucker. 1991. Semantics of pointers, referencing and dereferencing with intensional logic. In *LICS '91: Proceedings of the 6th symposium on logic in computer science*, 127–136. Washington, DC: IEEE Computer Society Press.

Jacobson, Pauline. 1999. Towards a variable-free semantics. *Linguistics and Philosophy* 22(2):117–184.

———. 2000. Paycheck pronouns, Bach-Peters sentences, and variable-free semantics. *Natural Language Semantics* 8(2):77–155.

Jäger, Gerhard. 2001. Anaphora and type logical grammar. Habilitationsschrift, Humboldt University Berlin. UIL-OTS Working Papers 01004-CL/TL, Utrecht Institute of Linguistics (OTS), Utrecht University.

Janssen, Theo M. V. 1997. Compositionality. In *Handbook of logic and language*, ed. Johan F. A. K. van Benthem and Alice G. B. ter Meulen. Amsterdam: Elsevier Science.

Kamp, Hans. 1981. A theory of truth and semantic representation. In *Formal methods in the study of language: Proceedings of the 3rd Amsterdam Colloquium*, ed. Jeroen A. G. Groenendijk, Theo M. V. Janssen, and Martin B. J. Stokhof, 277–322. Amsterdam: Mathematisch Centrum.

Kandulski, Maciej. 1988. The equivalence of nonassociative Lambek categorial grammars and context-free grammars. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 34(1):41–52.

Karttunen, Lauri. 1977. Syntax and semantics of questions. *Linguistics and Philosophy* 1(1):3–44.

Keller, William R. 1988. Nested Cooper storage: The proper treatment of quantification in ordinary noun phrases. In *Natural language parsing and linguistic theories*, ed. Uwe Reyle and Christian Rohrer, 432–447. Dordrecht: Reidel.

Kiselyov, Oleg. 2004. Zipper in Scheme. Usenet posting to the `comp.lang.scheme` newsgroup; `http://okmij.org/ftp/Scheme/zipper-in-scheme.txt`.

———. 2005. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Tech. Rep. 611, Computer Science Department, Indiana University.

Krifka, Manfred. 1995. The semantics and pragmatics of polarity items. *Linguistic Analysis* 25:209–257.

———. 2001. For a structured meaning account of questions and answers. In *Audiatur vox sapientiae: A festschrift for Arnim von Stechow*, ed. Caroline Féry and Wolfgang Sternefeld, 287–319. Berlin: Akademie Verlag.

Kroch, Anthony S. 1974. The semantics of scope in English. Ph.D. thesis, Massachusetts Institute of Technology. Reprinted by New York: Garland, 1979.

Kuno, Susumu, and Jane J. Robinson. 1972. Multiple wh questions. *Linguistic Inquiry* 3:463–487.

Ladusaw, William A. 1979. Polarity sensitivity as inherent scope relations. Ph.D. thesis, Department of Linguistics, University of Massachusetts. Reprinted by New York: Garland, 1980.

Lambek, Joachim. 1958. The mathematics of sentence structure. *American Mathematical Monthly* 65(3):154–170.

Lappin, Shalom, and Wlodek Zadrozny. 2000. Compositionality, synonymy, and the systematic representation of meaning. e-Print cs.CL/0001006, arXiv.org. Submitted to *Linguistics and Philosophy*.

Linebarger, Marcia Christine. 1980. The grammar of negative polarity. Ph.D. thesis, Massachusetts Institute of Technology.

May, Robert. 1985. *Logical form: Its structure and derivation*. Cambridge: MIT Press.

Milner, Robin. 1996. Calculi for interaction. *Acta Informatica* 33(8):707–737.

Moggi, Eugenio. 1990. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

———. 1991. Notions of computation and monads. *Information and Computation*

93(1):55–92.

Montague, Richard. 1974a. *Formal philosophy: Selected papers of Richard Montague*, ed. Richmond Thomason. New Haven: Yale University Press.

———. 1974b. The proper treatment of quantification in ordinary English. In *Formal philosophy: Selected papers of Richard Montague*, ed. Richmond Thomason, 247–270. New Haven: Yale University Press.

Moortgat, Michael. 1988. *Categorial investigations: Logical and linguistic aspects of the Lambek calculus*. Dordrecht: Foris.

———. 1995. In situ binding: A modal analysis. In *Proceedings of the 10th Amsterdam Colloquium*, ed. Paul Dekker and Martin Stokhof, 539–549. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

———. 1996. Generalized quantification and discontinuous type constructors. In *Discontinuous constituency*, ed. Harry C. Bunt and Arthur van Horck, 181–207. Berlin: Mouton de Gruyter.

———. 1997. Categorial type logics. In *Handbook of logic and language*, ed. Johan F. A. K. van Benthem and Alice G. B. ter Meulen, chap. 2. Amsterdam: Elsevier Science.

———. 2000. Computational semantics: Lab sessions. `http://www.let.uu.nl/~Michael.Moortgat/personal/Courses/cslab2000s.pdf`.

Moot, Richard. 2002. Proof nets for linguistic analysis. Ph.D. thesis, Utrecht Institute of Linguistics (OTS), Utrecht University.

Moran, Douglas B. 1988. Quantifier scoping in the SRI core language engine. In *Proceedings of the 26th annual meeting of the Association for Computational Linguistics*, 33–40. Somerset, NJ: Association for Computational Linguistics.

Morrill, Glyn V. 1994. *Type logical grammar: Categorial logic of signs*. Dordrecht: Kluwer.

———. 2000. Type-logical anaphora. Report de Recerca LSI-00-77-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.

———. 2003. On bound anaphora in type logical grammar. In *Resource-sensitivity in anaphora and binding*, ed. Geert-Jan M. Kruijff and Richard T. Oehrle, chap. 6. Dordrecht: Kluwer.

Nelken, Rani, and Nissim Francez. 2000. A calculus of interrogatives based on their algebraic semantics. In *Proceedings of TWLT16/AMILP2000: Algebraic methods in language processing*, ed. Dirk Heylen, Anton Nijholt, and Giuseppe Scollo, 143–160.

———. 2002. Bilattices and the semantics of natural language questions. *Linguistics and Philosophy* 25(1):37–64.

Nelken, Rani, and Chung-chieh Shan. 2004. A logic of interrogation should be internalized in a modal logic for knowledge. In *Proceedings from Semantics and Linguistic Theory XIV*, ed. Kazuha Watanabe and Robert B. Young,

197–211. Ithaca: Cornell University Press.

O'Neil, John. 1993. A unified analysis of superiority, crossover, and scope. In *Harvard working papers in linguistics*, ed. Höskuldur Thráinsson, Samuel D. Epstein, and Susumu Kuno, vol. 3, 128–136. Cambridge: Harvard University.

Pentus, Mati. 1993. Lambek grammars are context free. In *LICS '93: Proceedings of the 8th symposium on logic in computer science*, 429–433. Washington, DC: IEEE Computer Society Press.

———. 1996. Lambek calculus and formal grammars. Ph.D. thesis, Moscow State University, Moscow. In Russian. English translation: Pentus, Mati. 1999. Lambek calculus and formal grammars. In *Provability, complexity, grammars*, vol. 192 of *American Mathematical Society Translations Series 2*, 57–86. Providence: American Mathematical Society.

———. 1997. Product-free Lambek calculus and context-free grammars. *Journal of Symbolic Logic* 62(2):648–660.

Pierce, Benjamin C. 2002. *Types and programming languages*. Cambridge: MIT Press.

Plotkin, Gordon D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1(2):125–159.

Postal, Paul Martin. 1971. *Cross-over phenomena*. New York: Holt, Rinehart and Winston.

Queinnec, Christian, and Bernard Serpette. 1991. A dynamic extent control operator for partial continuations. In *POPL '91: Conference record of the annual ACM symposium on principles of programming languages*, 174–184. New York: ACM Press.

Quine, Willard Van Orman. 1960. *Word and object*. Cambridge: MIT Press.

Ranta, Aarne. 1994. *Type-theoretical grammar*. New York: Oxford University Press.

Reinhart, Tanya. 1983. *Anaphora and semantic interpretation*. London: Croom Helm.

Restall, Greg. 2000. *An introduction to substructural logics*. London: Routledge.

Reynolds, John C. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM national conference*, vol. 2, 717–740. New York: ACM Press. Reprinted as: Reynolds, John C. 1998. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation* 11(4):363–397.

———. 1983. Types, abstraction and parametric polymorphism. In *Information processing 83: Proceedings of the IFIP 9th world computer congress*, ed. R. E. A. Mason, 513–523. Amsterdam: Elsevier Science.

———. 1993. The discoveries of continuations. *Lisp and Symbolic Computation* 6(3–4):233–247.

Sabry, Amr. 1994. The formal relationship between direct and continuation-

passing style optimizing compilers: A synthesis of two paradigms. Ph.D. thesis, Department of Computer Science, Rice University. Also as Tech. Rep. TR94-241.

———. 1996. Note on axiomatizing the semantics of control operators. Tech. Rep. CIS-TR-96-03, Department of Computer and Information Science, University of Oregon.

Sabry, Amr, and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* 6(3–4):289–360.

Selinger, Peter. 2001. Control categories and duality: On the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science* 11: 207–260.

Shan, Chung-chieh. 2001. Monads for natural language semantics. In *Proceedings of the ESSLLI-2001 student session*, ed. Kristina Striegnitz, 285–298. Helsinki: 13th European Summer School in Logic, Language and Information.

———. 2004a. Delimited continuations in natural language: Quantification and polarity sensitivity. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke, 55–64. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.

———. 2004b. Polarity sensitivity and evaluation order in type-logical grammar. In *Proceedings of the 2004 human language technology conference of the North American chapter of the Association for Computational Linguistics*, ed. Susan Dumais, Daniel Marcu, and Salim Roukos, vol. 2, 129–132. Somerset, NJ: Association for Computational Linguistics.

———. 2004c. Shift to control. In *Proceedings of the 5th workshop on Scheme and functional programming*, ed. Olin Shivers and Oscar Waddell, 99–107. Tech. Rep. 600, Computer Science Department, Indiana University.

———. 2005. Linguistic side effects. In *Direct compositionality*, ed. Chris Barker and Pauline Jacobson. New York: Oxford University Press. In press.

Shan, Chung-chieh, and Chris Barker. 2005. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy*. In press.

Shieber, Stuart M. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy* 8:333–343.

Sitaram, Dorai. 1993. Handling control. In *PLDI '93: Proceedings of the ACM conference on programming language design and implementation*, vol. 28(6) of *ACM SIGPLAN Notices*, 147–155. New York: ACM Press.

Sitaram, Dorai, and Matthias Felleisen. 1990. Control delimiters and their hierarchies. *Lisp and Symbolic Computation* 3(1):67–99.

Sleator, Daniel Dominic, and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *Journal of the ACM* 32(3):652–686.

Søndergaard, Harald, and Peter Sestoft. 1990. Referential transparency, definiteness and unfoldability. *Acta Informatica* 27:505–517.

———. 1992. Non-determinism in functional languages. *The Computer Journal* 35(5):514–523.

Strachey, Christopher, and Christopher P. Wadsworth. 1974. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University Computing Laboratory. Reprinted as: Strachey, Christopher, and Christopher P. Wadsworth. 2000. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation* 13(1–2):135–152.

Szabolcsi, Anna. 1989. Bound variables in syntax (are there any?). In *Semantics and contextual expression*, ed. Renate Bartsch, Johan F. A. K. van Benthem, and Peter van Emde Boas, 295–318. Dordrecht: Foris.

———. 1992. Combinatory grammar and projection from the lexicon. In *Lexical matters*, ed. Ivan A. Sag and Anna Szabolcsi, 241–269. CSLI Lecture Notes 24, Stanford, CA: Center for the Study of Language and Information.

———. 1997. Strategies for scope taking. In *Ways of scope taking*, ed. Anna Szabolcsi, chap. 4, 109–154. Dordrecht: Kluwer.

———. 2004. Positive polarity—negative polarity. *Natural Language and Linguistic Theory* 22(2):409–452.

Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 26–37. New York: ACM Press.

Trueswell, John C., and Michael K. Tanenhaus, eds. 2005. *Approaches to studying world-situated language use: Bridging the language-as-product and language-as-action traditions*. Cambridge: MIT Press.

Wadler, Philip L. 1992a. Comprehending monads. *Mathematical Structures in Computer Science* 2(4):461–493.

———. 1992b. The essence of functional programming. In *POPL '92: Conference record of the annual ACM symposium on principles of programming languages*, 1–14. New York: ACM Press.

———. 1997. How to declare an imperative. *ACM Computing Surveys* 29(3): 240–263.

———. 2000. Proofs are programs: 19th century logic and 21st century computing. `http://homepages.inf.ed.ac.uk/wadler/topics/history.html`.

———. 2003. Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the ACM international conference on functional programming*. New York: ACM Press.

Zadrozny, Wlodek. 1994. From compositional to systematic semantics. *Linguistics and Philosophy* 17(4):329–342.