Effect-driven interpretation

Functors for natural language composition

Elements in Semantics

DOI: 10.xxxx/xxxxxxxx (do not change)
First published online: MMM dd YYYY (do not change)

Dylan Bumford
University of California, Los Angles

Simon Charlow
Rutgers University

Abstract: Computer programs are often factored into pure components—simple, total functions from inputs to outputs—and components that may have side effects—errors, changes to memory, parallel threads, abortion of the current command, etc. In this course, we make the case that human languages are similarly organized around the give and pull of pure and effectful processes, and we'll aim to show how denotational techniques from computer science can be leveraged to support elegant and illuminating semantic analyses of natural language phenomena.

Keywords: Semantics, Composition, Parsing, Functors, Monads, Scope

JEL classifications: A12, B34, C56, D78, E90

ISBNs: xxxxxxxxxxxxx(PB) xxxxxxxxxxxxx(OC)
ISSNs: xxxx-xxxx (online) xxxx-xxxx (print)

Contents

1 Introducing effects

1.1 Type-driven compositional semantics

Contemporary natural language semantic theories are generally drafted around three theoretically load-bearing components:

- a **syntax**, detailing the arrangement of grammatical expressions
- a **lexicon**, spelling out the meanings of individual expressions
- a theory of **composition**, describing how the meanings of complex expressions are built from their parts

Perhaps the simplest commonly-practiced architecture holds that expressions are constituted into binary-branching trees with lexical items at the leaves. Composition is governed by a simple theory of **types**, in the sense of Church (1940). A simple type is either primitive, corresponding to some basic sort of object (an entity `e`, an event `v`, a truth value `t`, etc.), or is constructed from two other types with an arrow $\tau_1 \to \tau_2$, corresponding to a function with domain τ_1 and codomain τ_2.

These types regulate an inventory of combinators, or **modes of combination**, which determine the range of ways that two meaningful expressions may combine. Given the simple type theory, by far the most common choices are forward and backward function application. For small, extensional fragments of language, this is often all that is required. But one may find appeal to other modes, including for example, various flavors of function composition (Ades & Steedman, 1982; Dowty, 1988), set restriction (Chung & Ladusaw, 2003; Kratzer, 1996), and set intersection (Kamp, 1975; Siegel, 1976).

With a grammar and a bank of combinatory modes in hand, composition is then said to be **type-driven** (Klein & Sag, 1985). The types of two daughter nodes are matched against the possible modes of combination. If the left type instantiates a combinator's first argument, and the right type its second, then the denotations of the daughters are composed as the combinator dictates. This basic setup is illustrated in Figure 1.

Given that types are used to dispatch modes of semantic combination, there must be an airtight correspondence between an expression's type and its denotation for this compositional regimen to make any sense. An expression of type $\mathtt{e} \to \mathtt{t}$ must in fact denote a function whose domain is the set of ordinary entities and whose codomain is the set of truth values. And an expression which denotes such a property of entities must in fact have type $\mathtt{e} \to \mathtt{t}$. In this sense, the semantics is said to be **strongly typed**, or type-safe.

Types:

$\tau ::= \mathtt{e} \mid \mathtt{t} \mid \ldots$ Primitive types

$\mid \tau \to \tau$ Function types

Modes of combination:

$(\mathtt{>}) :: (\alpha \to \beta) \to \alpha \to \beta$ Forward Application

$f \mathtt{>} x \coloneqq f\,x$

$(\mathtt{<}) :: \alpha \to (\alpha \to \beta) \to \beta$ Backward Application

$x \mathtt{<} f \coloneqq f\,x$

$(\circ) :: (\beta \to \varsigma) \to (\alpha \to \beta) \to \alpha \to \varsigma$ Backward Composition

$f \circ g \coloneqq \lambda x.\, f\,(g\,x)$

$(\sqcap) :: (\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{e} \to \mathtt{t}$ Predicate Modification

$f \sqcap g \coloneqq \lambda x.\, f\,x \wedge g\,x$

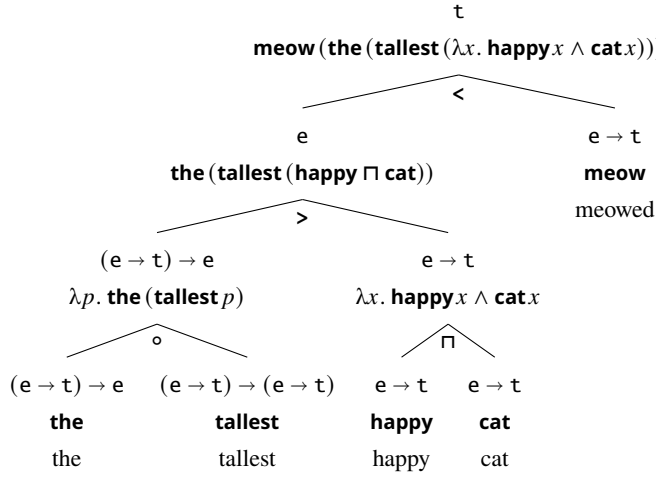
$(\restriction) :: (\alpha \to \beta \to \mathtt{t}) \to (\alpha \to \mathtt{t}) \to \alpha \to \beta \to \mathtt{t}$ Relation Restriction

$r \restriction p \coloneqq \lambda x \lambda y.\, r\,x\,y \wedge p\,x$

Figure 1 A simple type-driven grammar

Throughout this Element, we will display type-driven derivations as trees recording the types of the constituents derived. Below each branching node in a derivation tree, we will identify the mode of combination used to combine that node's daughters. An example, using some of the combinators in Figure 1, is given in (1.1). Sometimes, as in (1.1), we will also provide reduced lambda terms below the types of nodes, corresponding to their denotations, with constants standing for the denotations of lexical items printed in **bold**. However, denotations are always recoverable by recursively applying the combinators below nodes to the denotations of their daughters, so they are generally omitted except where we think they may be clarifying.

(1.1)



As it happens, the more or less canonical combinators in Figure 1 have the property that at most one of them will apply to any given pair of daughters. This guarantees that interpretation is **deterministic**: for any specific syntactic structure, there will be at most one interpretation (though there may of course be more than one well-typed structure for a given string). In principle, however, with a larger or different inventory of modes of combination, there could be a node at which more than one combinatory rule is applicable. In this case, the types would underdetermine the meaning of the complex expression, predicting only *a set* of possible interpretations. Indeed, we will at many places in this Element make use of this compositional indeterminacy in the prediction of various systematic ambiguities.

1.2 More than just a name

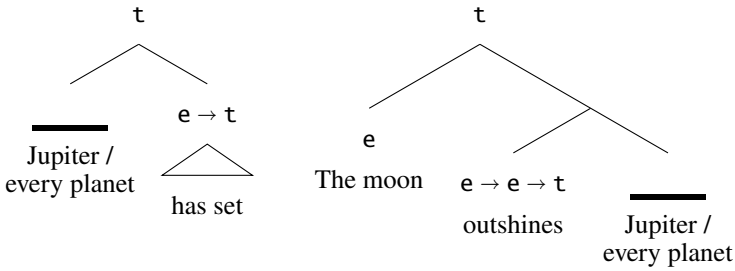
One consequence of the picture in Figure 1 is that generally speaking, when two expressions have the same syntactic distribution, they must also have the same type. As every student of semantics knows, this leads immediately to trouble. Most famously, quantificational noun phrases by and large occur in the same syntactic positions as proper names. For instance, like names, they are just as felicitous in subject positions as they are in object positions.

(1.2a) {Jupiter, every planet} has set

(1.2b) The moon outshines {Jupiter, every planet}

But the only type that can be combined in both of the trees of (1.3) is e.

(1.3)



For proper names this is sensible, of course. But for quantificational phrases it is absurd. Well-rehearsed entailment patterns show that there can be no singular entity that is the referent of 'every planet', or 'no planet', or 'at least one planet', etc. And given the required type-safety of the semantics, they therefore cannot be expressions of type e.

Solutions to this problem break in every conceivable direction: reject that these are the trees that are interpreted, reject that these are the correct types of the lexical items, reject that these are the only modes of combination, reject the type system, and so on. But one perhaps underappreciated aspect of the compositional conundrum laid bare in (1.3) is that it is not at all specific to matters of quantification. The basic nature of the problem is that an expression, say 'every planet', appears to play exactly the same argument-structural role as a name, yet clearly contributes something other than a simple referent to the meaning of the sentence. One might say the same is true of 'wh'-expressions, or indefinites, or disjunctions, as in (1.4). These are not (obviously) quantifiers, but they are (obviously) not names either. The arguments they provide to their predicates are *indeterminate*, just a stock of potential witnesses conjured up in

parallel.

(1.4a) Which planet is next to the moon

(1.4b) A planet is next to the moon

(1.4c) Jupiter or Mars is next to the moon

Definites too do not have the same semantic profile as names, despite saturating the same argument positions. What they refer to, or whether they refer at all, depends on what things are contextually salient. But an expression that does not have a stable individual referent cannot have type e. Pronouns, demonstratives, and indexicals even more clearly depend on context in a way that the type e does not represent.

(1.5a) The planet in the West is next to the moon

(1.5b) It is next to the moon

(1.5c) This planet that I'm looking at is next to the moon

In fact, with a bit of prosodic focus, any noun phrase can be made to contribute more to the meaning of the sentence than just its referent. The sentences in (1.6) do not have the same truth conditions; the first is true when nothing relevant is visible except for Jupiter's moon, and the second true when no other relevant moon is visible except for Jupiter's. The only difference between them is which phrase is focused, so the meanings of the focused phrases must somehow encode the necessary information.

(1.6a) Only [Jupiter's moon]$_{\mathbf{F}}$ is visible

(1.6b) Only [Jupiter]$_{\mathbf{F}}$'s moon is visible

Likewise, even more transparently, a noun phrase may always be supplemented by various sorts of parenthetical and appositive constructions. For instance, the supplemented subject of (1.7a) and object of (1.7b) clearly saturate the same argument positions as the plain name that anchors them. Yet the supplemented phrases obviously add propositional information to the sentence, information that cannot be found in any individual entity.

(1.7a) Mars, which is now next to the moon, is setting

(1.7b) Jupiter has passed Mars, which is now next to the moon

Again, all of the phrases in (1.4)–(1.7) behave for compositional purposes

as if they denoted ordinary entities. They are, by and large, grammatical and sensible wherever something of type e would be grammatical and sensible. But they cannot, or at least can't merely, denote entities. Their semantics is necessarily more complicated than this.

And these kinds of discrepancies are by no means limited to noun phrases. There are adjunct 'wh'-expressions that modify the same sorts of properties as ordinary adjuncts, and disjunctions of every category. There are verbs that differ only in their presuppositions. There are pro-forms and gaps and traces over arbitrarily complex types, parentheticals that can attach to almost anything, and of course quantifiers galore. All over the place we see expressions with interesting and complicated semantic properties that are nevertheless squeezed into positions where semantically boring expressions are expected, and this does not seem to disrupt composition in the slightest.

In this Element we adopt the view that these sorts of enriched expressions ought to be analogized to **impure** components of programming languages. This view has many precedents. Chief among them are analyses in the dynamic bent of Heim (1982), especially as described by Groenendijk and Stokhof (1991), Muskens (1990, 1996), Eijck (2001), and colleagues, as well as analyses making heavy use of continuation-passing, including de Groote (2001), Barker (2002), Barker and Shan (2014), and Kiselyov and Shan (2014). The perspective we present here seeks to unify some of this work and much independent work on composition in various empirical domains.

In so doing, we largely follow the program introduced to linguistics by Shan (2001a, 2005), and built upon by Charlow (2014), Asudeh and Giorgolo (2020), and others. This program borrows directly from a tradition in computer science incorporating concepts from Category Theory that isolate repeating algebraic patterns that arise when working with various mathematical structures. These patterns have been fruitfully applied in explicating the semantics of common programming language constructs, and also in streamlining the design of type-safe programming languages themselves.

As linguists, we benefit from this work on both ends. On the one hand, natural languages are much like *weakly-typed* programs. As we've already sketched, all sorts of weird semantic stuff can happen that the grammatical system, the compiler if you will, is blind to. In that sense, the mathematical concepts undergirding compositional semantic descriptions of such semantically devious languages can be imported, often wholesale, to linguistics. On the other hand, the strongly-typed programming languages engineered specifically to avoid any semantic shenanigans can be used as an executable formalism for implementing compositional analyses of languages of the former, screwy, variety, including the ones we use every day.

We will not presuppose any particular exposure to programming or programming language theory. The entire discussion is intended to be accessible to anyone familiar with the usual aims and methods of compositional natural language semantics. But we will also include snippets of code implementing key ideas, if for no other reason than because it is so easy to do so, and it makes for a much more engaging and robust means of theorizing.

1.3 Effects in programming languages

The constructs of so-called **imperative** programming languages are often divided into two categories. On the one hand there are **pure** expressions that carry out the basic business of determining a concrete result, or **value**. These include **literal** expressions like characters, strings, numbers, and booleans, whose values are fixed and stable, what we might call the names of the language. Other pure expressions include the (total) functions that convert one value to another. Usually a language provides a library of such functions, including straightforward boolean and arithmetic operations, like negation and addition and such. These we might think of as the ordinary nouns and verbs and adjectives of the language. Semantically, they are no more interesting than their graphs, the pairings between their inputs and outputs.

On the other hand there are **statements**. These are the bits of program text that determine the evaluation order and control flow of a computation, that is, what happens when the program is **executed**. Where pure expressions are intended to refer or otherwise reduce to some or other plain value, statements are intended *do things*, like change addresses in memory, (re-)assign and/or allocate variables, throw errors, spawn threads, start loops, and print things to your screen. These sorts of non-referential processes are loosely referred to as **(side) effects** of computation.

As in natural language, program snippets containing commands like these may appear in the same places that pure program snippets would. For instance, the function `plus` on the left of Figure 2 is pure. It simply returns the sum of its two inputs. The function `showplus` on the right is impure. It also returns the sum of its two inputs, but additionally prints some words to the screen. In any given calling context, like `5 * ___`, the two functions will yield the same value. But when the one on the right is executed, the additional words are printed, as displayed in Figure 2.

How then do programming language theorists think about the meanings of programs like `showplus`? The answers to that question are certainly no less varied and creative than the answers that linguists have given to the issues

```
function plus(x,y) = {

  return x + y
}

> 5 * plus(3, 7)

50
```

```
function showplus(x,y) = {
  console.log("doing (+)");
  return x + y
}

> 5 * showplus(3, 7)
"doing (+)"
50
```

Figure 2 Pure and impure javascript programs

of composing the phrases introduced above. But, excitingly, they are often different! In this Element, we do not pretend to offer a survey of the semantic methods deployed by computer scientists for reasoning about effects. Instead we concentrate on a few of the algebraic and combinatorial techniques that have proved useful in linguistic theorizing.

The first order of business is rectifying the situation with the types. Type-safety requires that any semantically-relevant behavior of a program be reflected in its type. The same is true in language. To model the variety of behaviors on display in (1.4)–(1.7), it will be helpful to first expand the type system.

1.4 Algebraic Data Types

Some of the exhibited linguistic effects seem to call for denotations with *multiple dimensions* of meaning. The natural mathematical setting for modeling multi-dimensionality is a tuple, with different semantic dimensions in different coordinates. We thus introduce **product types** $\alpha \times \beta$ to model meanings that carry both type α and type β content. Denotationally, an expression of type $\alpha \times \beta$ takes its meaning from the Cartesian product of α and β.

(1.8) Mars, a planet :: $\mathtt{e} \times \mathtt{t}$

$[\![\text{Mars, a planet}]\!] = \langle \mathbf{m}, \mathbf{planet\ m} \rangle$

Other effects seem to call for denotations with *multiple variants* of meaning. For instance, a definite description will either refer to an object (type $\mathtt{e}$), or it will fail to refer, returning a computational dead end. We might model a failure of reference with a type $\perp$ whose only value is #. The description therefore denotes a computation that will return a value of one of two distinct types, either

type $\mathtt{e}$ or type $\bot$. The type of such a computation is naturally modeled by a **sum type** $\alpha + \beta$. Denotationally, an expression of type $\alpha + \beta$ takes its meaning from the disjoint union of α and β.

(1.9) the planet :: $\mathtt{e} + \bot$
$[\![\text{the planet}]\!] = x$ if **planet** $= \{x\}$ else #

Types built from product, sum, and arrow constructors are called **Algebraic Data Types**. Finally, we will want to be able to define the types of values that are drawn from *powersets* of specific domains. For this we will slightly abuse the notation $\{\alpha\}$ to represent the type whose members are sets of type-α things. For instance, the canonical Hamblin denotation for a 'wh'-argument is a set of entities (Hamblin, 1973), and its type therefore is $\{\mathtt{e}\}$.

(1.10) which planet :: $\{\mathtt{e}\}$
$[\![\text{which planet}]\!] = \{x \mid \textbf{planet}\, x\}$

With these algebraic data types as scaffolding for structured values, the entries in Table 1 spell out some more or less standard semantic characterizations of the constructions in (1.4)–(1.7). Of course there are many competing analyses for each of these expressions, but we won't pause to motivate the choices in Table 1. The entries are intended primarily as illustrative test cases for the approach to compositionality we will describe.

The first four rows are the examples we've seen, deploying the complex types that represent the different kinds of structured values we will make use of. Pronouns and the like are naturally construed as functions from some sort of context $\mathtt{r}$ to a referent. Different theories of anaphora resolution may choose to model linguistic contexts in different ways (and so choose different domains for the type $\mathtt{r}$ and different selection functions $(\cdot)_0$), but as far as we are aware, every semantics for pronominal/indexical elements has this basic functional shape. Definite descriptions denote "partial entities"; they either succeed in referring to an entity or they result in failure. Supplemented names are bidimensional, including both an ordinary referent and a fact about that referent. And 'wh'-expressions refer indeterminately, denoting the set of all their possible answers.

The next four rows in the table are slightly more complex, involving nested constructions of types. Quantificational phrases denote Generalized Quantifiers, properties of properties. Focused phrases are both bidimensional and indeterminate; a certain entity is named, while the alternatives to that entity are evoked. And then we list a topic-marked name and an indefinite in the

Expression	Type	Denotation
it	$\mathtt{r} \to \mathtt{e}$	$\lambda g.\, g_0$
the planet	$\mathtt{e} + \bot$	x if $\mathbf{planet} = \{x\}$ else #
Jupiter, a planet	$\mathtt{e} \times \mathtt{t}$	$\langle \mathbf{j}, \mathbf{planet}\mathbf{j} \rangle$
which planet	$\{\mathtt{e}\}$	$\{x \mid \mathbf{planet}x\}$
no planet	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$	$\lambda c.\, \neg\exists x.\, \mathbf{planet}x \wedge c\,x$
$\text{Jupiter}_{\mathbf{F}}$	$\mathtt{e} \times \{\mathtt{e}\}$	$\langle \mathbf{j}, \{x \mid x \in D_{\mathtt{e}}\} \rangle$
as for Jupiter	$\mathtt{s} \to (\mathtt{e} \times \mathtt{s})$	$\lambda s.\, \langle \mathbf{j}, \mathbf{j} + s \rangle$
a planet	$\mathtt{s} \to \{\mathtt{e} \times \mathtt{s}\}$	$\lambda s.\, \{\langle x, x + s \rangle \mid \mathbf{planet}x\}$

Table 1 Example noun phrases, their types and denotations

style of dynamic semantics. Topics, almost by definition, not only refer, but also put their referent front and center on some sort of evolving discourse stage. This is naturally modeled by the deterministic update in the table, which both holds out **j** for composition (as in the appositive and focus rows) and also rotates **j** to the front of a context s. Typical dynamic analyses of indefinites are similar, in that they change the conversational state so that their witnesses are available for anaphora. But like 'wh'-expressions, indefinites do not necessarily single out unique referents, and so in general correspond to updates that are **nondeterministic**. Again, theories may differ in how they model discourse contexts (determining different domains for $\mathtt{s}$ and different update functions $+$), but all dynamic semantics in the wake of Heim (1982) have the basic relational update procedure from inputs s to modified outputs $s + x$.

The perspective we would like to encourage here views these expressions as denoting particular kinds of **computations**, specifically computations that yield entities. A pronoun reads a referent off of an environment. An appositive writes a fact to the common ground. A 'wh'-expression introduces a slate of choices that fork the sentence into several parallel threads. A definite description is a program that might crash if executed in the wrong situation. An indefinite modifies what referents are in memory, and possibly what addresses they're stored in. And so on.

Remember, all of these expressions appear in positions where ordinary entities are expected. And naturally their denotations all, one way or another, contain, return, manipulate, abstract over, or quantify over entities. Following the programming literature, we will sometimes talk about the entities in these

types as being situated in a particular **computational context**, and use the term **effect** to refer to whatever a computation does with them.

To make this formal, let us introduce *ad-hoc* type constructors for each of the effects in Table 1. Here we use the term (unary) **type constructor** to mean a function from types to types. We find the constructor for each effect by abstracting over `e`. For instance, the `G` constructor encodes the effect of reading from an environment of type `r`. The `W` constructor encodes the effect of logging a message of type `t`. The `M` constructor the effect of possibly failing.

(1.11)
$$\begin{aligned} \mathtt{G}\,\alpha &::= \mathtt{r} \to \alpha & \mathtt{C}\,\alpha &::= (\alpha \to \mathtt{t}) \to \mathtt{t} \\ \mathtt{W}\,\alpha &::= \alpha \times \mathtt{t} & \mathtt{F}\,\alpha &::= \alpha \times \{\alpha\} \\ \mathtt{M}\,\alpha &::= \alpha + \bot & \mathtt{S}\,\alpha &::= \{\alpha\} \\ \mathtt{T}\,\alpha &::= \mathtt{s} \to (\alpha \times \mathtt{s}) & \mathtt{D}\,\alpha &::= \mathtt{s} \to \{\alpha \times \mathtt{s}\} \end{aligned}$$

Then we can express our dictionary of noun phrases as in Table 2. This makes clear that all of these expressions are entity-directed computations. The particular nature of each computation is encoded in the structure of the effect defined by its type constructor in (1.11).

Expression	Type	Denotation
it	`G e`	$\lambda g.\, g_0$
the planet	`M e`	x if **planet** $= \{x\}$ else #
Jupiter, a planet	`W e`	$\langle \mathbf{j}, \mathbf{planet}\,\mathbf{j} \rangle$
which planet	`S e`	$\{x \mid \mathbf{planet}\,x\}$
no planet	`C e`	$\lambda c.\, \neg\exists x.\, \mathbf{planet}\,x \wedge c\,x$
$\text{Jupiter}_{\mathbf{F}}$	`F e`	$\langle \mathbf{j}, \{x \mid x \in D_{\mathtt{e}}\} \rangle$
as for Jupiter	`T e`	$\lambda s.\, \langle \mathbf{j}, \mathbf{j} \mathbin{+\!\!+} s \rangle$
a planet	`D e`	$\lambda s.\, \{\langle x, x \mathbin{+\!\!+} s \rangle \mid \mathbf{planet}\,x\}$

Table 2 Example noun phrases, with types encoded by effect constructors

So far all we have done is relabel the types of different kinds of values. None of this provides any immediate relief to the problem at hand, which is to fit these expressions into contexts that only know how to process an ordinary entity. On the contrary, hiding all the mathematical structure of the types would seem to preclude rather than facilitate type-driven composition.

But it turns out that all of these constructors — `G`, ..., `D` — share a few

very important algebraic properties. And these properties allow us to ignore whatever is specific to the kind of computation represented by the type, and get on with the work of passing the underlying entity into the predicate that it saturates. This not only paves the way for composition, it also reveals a certain uniformity that is completely lost in all of the independent theorizing about the various linguistic phenomena. At the same time, it frees researchers working on independent semantic problems to concentrate on their effects of interest without inventing idiosyncratic mechanisms of scope, type-shifting, and combination.

In the coming chapters, we spell out some of the algebraic properties of these effects. But first, let us lay some computational groundwork for the discussion. As mentioned in Section 1.1, the compositional framework we will present out is categorematic and type-driven. That is, the ways of interpreting a constituent will depend only on the types of its daughters. Moreover, the rules determining which pairs of types lend themselves to which modes of combination are going to be entirely formal and decidable. You just have to look at the types and see if they match the rules.

In other words, it should all be stupid enough that a computer can do it for us. Let us then put our machine where our mouth is, and implement a very simple type-driven interpreter. We will do this in the programming language Haskell, whose construction and development has been heavily influenced by the algebraic concepts discussed here. Unfortunately space precludes any proper introduction to the constructs of Haskell, but the syntax was designed to mimic standard mathematical notation, and we hope the code will be quite readable even to those unfamiliar with the language.

1.5 Implementing a type-driven interpreter

To get the ball rolling, let's start with the type system in Figure 1, setting aside effects for now. Throughout this Element, we will build on this interpreter, folding in the combinatoric operations that we introduce as we go.

Our goal here is to have the computer figure out every way that a sentence can be interpreted, knowing only its constituency structure and the types of its lexical items. As such, we focus here on the process of type-driven combination, rather than any aspect of parsing. We will assume then that the sentences to be interpreted are pre-assembled into constituents with typed leaves. Here is a Haskell data type representing such parsed structures.

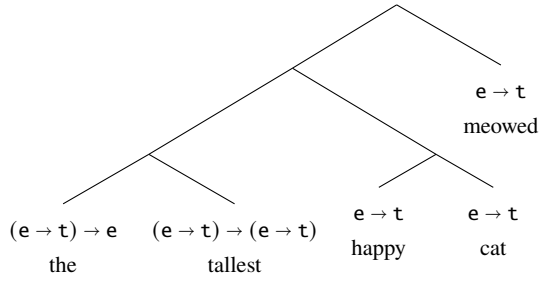
```
data Syn
  = Leaf Ty String
  | Branch Syn Syn

data Ty
  = E | T       -- primitive types
  | Ty :-> Ty   -- function types
```

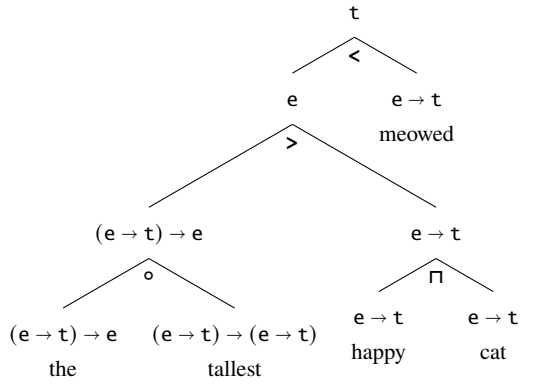
A piece of syntax `Syn` is either a `Leaf` containing a `String` (the name of the lexical item) and a `Ty` (its type), or it's a `Branch`ing node containing two subtrees. A type `Ty` is either atomic — `E` or `T` — or complex — an arrow between two other types.

The goal, again, is to determine the possible modes of combination for each node and what the resulting type would be. It is, in effect, to induce the tree in (1.12b) from that in (1.12a).

(1.12a)



(1.12b)



To do this, we define a data structure for type- and mode-annotated trees, which we call `Sem` trees, analogous to the `Syn` trees defined above. A `Sem` object is either a typed lexical item or a mode of combination together with two daughters (themselves type- and mode-annotated trees). A mode of combination `Mode` is, for now, just a tag indicating which of the modes from Figure 1 applies. They are just Haskell versions of the symbols in (1.12b).

```
data Sem
  = Lex Ty String
  | Comb Ty Mode Sem Sem

data Mode
  = FA -- forward application
  | BA -- backward application
  | PM -- predicate modification
  | BC -- backward composition
       -- other basic modes of combination, as desired
```

The engine of the type-driven logic is implemented by the function `modes`. All it does is pattern-match on the two types that it is handed. If the left type is an arrow type `a :-> b` and the right type is `a`, then `FA` is an applicable mode of combination, and the result will be of type `b`. Vice versa for `BA`. If both inputs are arrow types with co-domain `T` and identical domain types, then `PM` applies. And so on.

```
modes :: Ty -> Ty -> [(Mode, Ty)]
modes l r = case (l, r) of
  (a :-> b, _      ) | r == a -> [(FA, b)]
  (_      , a :-> b) | l == a -> [(BA, b)]
  (E :-> T, E :-> T)          -> [(PM, E :-> T)]
  (c :-> d, a :-> b) | b == c -> [(BC, a :-> d)]
  --       ...           ==    ...
  _                           -> []
```

Notice that result of applying `modes` to two types `l` and `r` is a *list* of possible results. If any of the substantive cases match, the result is a singleton list containing just the mode appropriate to that case. If none of them match, the result is an empty list. So the result of any call of `modes l r` is at most a singleton list. Thus the interpreter is deterministic. In later chapters, this will not be the case. The result of combination may contain 0, 1, or more modes and the corresponding result types.

Finally, we define the interpreter `synsem`. Given a syntactic object `Syn`, it returns a list of possible type- and mode-annotations: `[Sem]`. It does this by a very straightforward recursion, bottoming out at the leaves. Given a branching node `Branch lsyn rsyn`, it interprets the left branch, interprets the right branch, and then combines the types of the results with `modes`. Because interpretation is relational rather than functional, each of these actions yields not one, but a list of intermediate results. The final collection of `Sem` annotations is determined by taking each possible interpretation `lsem` of the left branch, each possible interpretation `rsem` of the right branch, and each possible way of putting such interpretations together to yield a new `Comb`ined constituent.

```
synsem :: Syn -> [Sem]
synsem syn = case syn of
  (Leaf t w)          -> [Lex t w]
  (Branch lsyn rsyn) ->
    [ Comb ty op lsem rsem
      | lsem     <- synsem lsyn
      , rsem     <- synsem rsyn
      , (op, ty) <- modes (getType lsem) (getType rsem) ]
  where
    getType (Comb ty _ _ _) = ty
```

2 Functors

2.1 Maps and mapping

The computational contexts encoded in the types of Table 2 are quite varied, but they are all known to Category Theorists and computer scientists as **functors**. Intuitively speaking, a constructor θ is functorial if its parameter α remains accessible to manipulation despite being embedded in the θ structure. It should moreover not make any difference what kind of thing α is. Just knowing how it is situated inside the θ structure should be enough to know how it could be adjusted.

For instance, given a set of numbers S, and a function k to update those numbers, we can modify the members of S by **mapping** k over it, that is, applying it pointwise.

$$\begin{aligned} &S = \{1, 2, 3\} && S' = \{k\, n \mid n \in S\} \\ &k = \lambda n.\, n + 1 && = \{2, 3, 4\} \end{aligned} \tag{2.1}$$

We could do the same if k converted numbers to strings.

$$\begin{aligned} &S = \{1, 2, 3\} && S' = \{k\, n \mid n \in S\} \\ &k = \lambda n.\, \mathbf{repeat}\, n\, \mathbf{"a"} && = \{\mathbf{"a"}, \mathbf{"aa"}, \mathbf{"aaa"}\} \end{aligned} \tag{2.2}$$

Likewise if S were full of entities, and k a function from entities to Booleans.

$$\begin{aligned} &S = \{\mathbf{j}, \mathbf{m}, \mathbf{s}\} && S' = \{k\, x \mid x \in S\} \\ &k = \lambda x.\, x = \mathbf{j} && = \{\mathbf{T}, \mathbf{F}\} \end{aligned} \tag{2.3}$$

In every case, the way a function k looking for α-type argument is "applied" to a set S of α-type elements is the same: $S' = \{k\, a \mid a \in S\}$. Similarly, given a pair W whose left coordinate is a number, and a function k that updates numbers, we can map k over the pair by applying it just to the left coordinate.

$$\begin{aligned} &W = \langle 1,\ q \rangle && W' = \langle k\, W_0,\ W_1 \rangle \\ &k = \lambda n.\, n + 1 && = \langle 2,\ q \rangle \end{aligned} \tag{2.4}$$

And again, it's clear we could do the same thing no matter what the return type of k was, or indeed what kind of thing is sitting in the left coordinate. As long as that coordinate is type α and $k :: \alpha \to \beta$, this way of "applying" k to W will be sensible.

Formally, a type constructor θ is a functor if there is an operation $(\bullet)$ with the type indicated in (2.5) respecting the two laws in (2.6).

(2.5) $(\bullet) :: (\alpha \to \beta) \to \Theta\,\alpha \to \Theta\,\beta$

(2.6) **Identity:** $\mathbf{id} \bullet M = M$

Composition: $f \bullet (g \bullet M) = (f \circ g) \bullet M$

That is, for an operation $(\bullet)$ to count as a **map**, it should not interact in any way with the effect structure of Θ, concentrating all its energy on applying k to the embedded α. So mapping the identity function **id** over a structure M should not change anything at all about M. This is because **id** does not change the α element(s) in M, and $(\bullet)$ does not change the non-α elements in M or the structure of M itself. And mapping should be a homomorphism with respect to function composition; it shouldn't matter whether you map a composite operation $f \circ g$ over M, or first map g over M and then f over the result. As it happens, whenever $(\bullet)$ is defined by a **polymorphic** lambda term, the first law entails the second (Wadler, 1989).

In Haskell, the $(\bullet)$ operation is known as `fmap`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

For many type constructors Θ, the relevant mapping operation is utterly straightforward, as often there is only one possible parametric function $(\bullet) :: (\mathtt{a} \to \mathtt{b}) \to \Theta\,\mathtt{a} \to \Theta\,\mathtt{b}$. Indeed any constructor whose parameter instances α are all in **positive** positions is guaranteed to be a functor, and its $(\bullet)$ mechanically derivable. As it happens, this includes all of the examples in Table 2.

In fact, many of these effects are so essential to structuring programs that they have canonical names in Haskell, defined more or less as in Table 3. The functor instances of these constructors are defined in the standard prelude or in standard libraries. We give simplified versions of these definitions in Appendix A2.

As seen in the table, many of the corresponding Haskell constructors have additional type parameters. For instance, where we write $\mathtt{G}\,\alpha ::= \mathtt{r} \to \alpha$, the corresponding Haskell constructor is declared as `data Reader r a = Reader (r -> a)`. The `r` here corresponds to the type of the environment that anaphoric computations read from. This environment can take a great variety of forms, and any particular computation will need to specify which form is assumed. Likewise, the `W`/`Writer` effect is parameterized by what kind of values are stored in the supplemental dimension, and the `S`/`State` effect by how discourse contexts are encoded, and so on. In formal presentations, we will leave these parameters implicit to avoid typographic clutter, since these choices mostly do not matter for our purposes. When we do make particular choices, we will discuss the

Mathematical Type	Haskell Type
$\mathtt{G}\,\alpha ::= \mathtt{r} \to \alpha$	`data Reader r a = Reader (r -> a)`
$\mathtt{W}\,\alpha ::= \alpha \times \mathtt{t}$	`data Writer t a = Writer (a, t)`
$\mathtt{M}\,\alpha ::= \alpha + \bot$	`data Maybe a = Just a | Nothing`
$\mathtt{S}\,\alpha ::= \{\alpha\}$	`type S a = [a] -- our S constructor approximates -- the built-in circumfix list constructor []`
$\mathtt{T}\,\alpha ::= \mathtt{s} \to (\alpha \times \mathtt{s})$	`data State s a = State (s -> (a,s))`
$\mathtt{C}\,\alpha ::= (\alpha \to \mathtt{t}) \to \mathtt{t}$	`data Cont t a = Cont ((a -> t) -> t)`

Table 3 Haskell conventions for common effect types

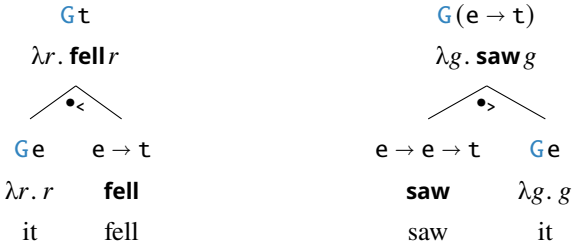
relevant types in surrounding prose.

2.1.1 Mapping as a mode of combination

How does any of this help with problems of composition stressed in Chapter 1? No doubt the simplest thing to do would be to add (•) to the inventory of combinatory modes, perhaps a forward version and a backward version in analogy with ordinary forward and backward Function Application.

Such combinators, defined in Figure 3, provide for derivations like those in (2.7). Both of the derived constituents in (2.7) combine a pronoun with an ordinary predicate of entities. To keep things quite simple at the start, let us adopt a **variable-free** view of anaphora, wherein the computations denoted by pronouns are mere requests for antecedents, nothing more (see, e.g., P. Jacobson 1999; P. I. Jacobson 2014, et seq.). In this conception, anaphora resolution is the process of choosing how such requests should be fulfilled (what values to pass in for the open arguments), but the pronouns themselves do no semantic selectional work. That is, a pronoun's job is just to make the request, and then to hand over the antecedent it receives for further composition.

(2.7)



Combinators

$(>) :: (\alpha \to \beta) \to \alpha \to \beta$ — Forward Application

$f > x := f\,x$

$(<) :: \alpha \to (\alpha \to \beta) \to \beta$ — Backward Application

$x < f := f\,x$

$\vdots$ — Other Basic Combinators

$(\bullet_>) :: (\alpha \to \beta) \to \Theta\alpha \to \Theta\beta$ — Forward Map

$k \bullet_> X := k \bullet X$

$(\bullet_<) :: \Theta\alpha \to (\alpha \to \beta) \to \Theta\beta$ — Backward Map

$X \bullet_< k := k \bullet X$

Figure 3 Adding basic (•) combinators to the grammar

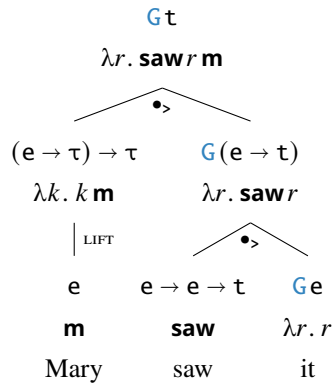
One nice aspect of the grammar in Figure 3 is that there is no substantive difference between the way effect-typed expressions compose in subject vs. object positions. As usual with natural language, a function may occur on either the left or right of its argument, but the semantics is the same in both cases. However, with just the combinatory inventory of Figure 3, it is not possible to combine a computational predicate — type $\mathsf{G}(\mathsf{e} \to \mathsf{t})$ — with an ordinary subject — e. This is because the mapping operation (•) always lifts an ordinary *function* over an effectful *argument* (as opposed to applying an effectful function to an ordinary argument). We could add further modes of combination that do this using (•), or we could allow ordinary arguments to be **lifted** into ordinary functions à la Partee (1986).

(2.8) $(\cdot)^{\text{LIFT}} :: \alpha \to (\alpha \to \tau) \to \tau$

$x^{\text{LIFT}} := \lambda k.\, k\,x$

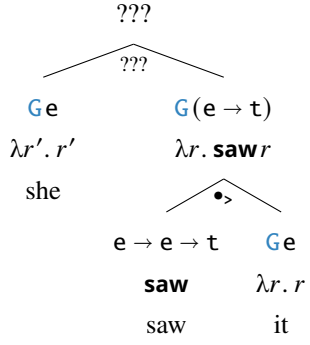
This is shown in (2.9).

(2.9)



The real trouble for this simple-minded approach to incorporating functoriality begins when an expression and its sister *both* denote computations. This situation is illustrated in (2.10). Even with free lifting, there is no way to put the subject and the predicate of (2.10) together.

(2.10)



2.2 Higher-order effects

The first step in redressing this unfortunate incomposability is deciding what sort of thing (2.10) ought to denote. The type of the subject, $\mathtt{G}\,\mathtt{e}$, indicates that it needs an antecedent. The type of the predicate, $\mathtt{G}(\mathtt{e} \to \mathtt{t})$, indicates that it also, independently, needs an antecedent. The denotation of the full sentence should honor both of these requests, so its type should indicate that it needs *two* antecedents.

(2.11) she saw it :: $\mathtt{G}(\mathtt{G}\,\mathtt{t})$

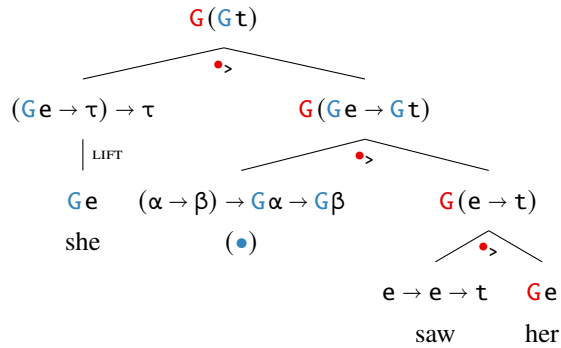
(2.12) $[\![\text{she saw it}]\!] = \lambda r \lambda r'.\ \mathbf{saw}\, r\, r'$

We show here two ways to use the functoriality of $\mathtt{G}$ to make this happen. Both take advantage of the following idea. Even though $\mathtt{G}\,\mathtt{e}$ and $\mathtt{G}(\mathtt{e} \to \mathtt{t})$ cannot be combined (because neither can be mapped over the other), the underlying type of the predicate, $\mathtt{e} \to \mathtt{t}$, could be combined with the *full* type of the subject $\mathtt{G}\,\mathtt{e}$ (as in (2.7)). And because $\mathtt{G}$ is a functor, we should be able to map this underlying mode of combination over the predicate's $\mathtt{G}$ shell. Since the mode of combination that combines the subject, $\mathtt{G}\,\mathtt{e}$, with the underlying type of the predicate, $\mathtt{e} \to \mathtt{t}$, is $(\bullet)$, what we need then is some way to map $(\bullet)$ itself over the daughters.

2.2.1 Mapping in the language

The most direct route to this sort of higher-order mapping is to add $(\bullet)$ to the object language. Then $(\bullet)$ might be mapped over a computation just the same as any other function. This is effectively the strategy that P. Jacobson (1999) adopts, though with combinators specific to $\mathtt{G}$ and none of our effect-oriented conceit.[1]

(2.14)



[1]The reader may wish to compare (2.14) to P. Jacobson's (1999: p. 139) Example (31), given in (2.13a), and transliterated to our notation in (2.13b) below. The translation proceeds by rewriting $\mathbf{g}_0$ as $(\bullet)$ and recognizing that P. Jacobson's $\mathbf{g}_n$ is equivalent to $\mathbf{g}_0\,\mathbf{g}_{n-1}$. Then in (2.13c), we write as many $(\bullet)$'s as possible as infix operators, yielding a logical form isomorphic to the tree in (2.14).

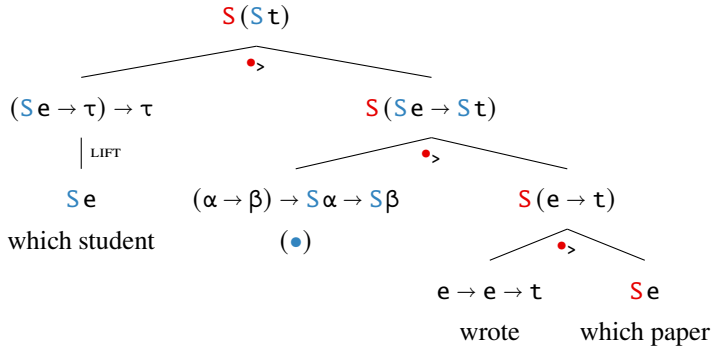
(2.13a) $\mathbf{g}_0$ (LIFT $[\![\text{his mother}]\!]$) ($\mathbf{g}_1$ ($\mathbf{g}_0$ $[\![\text{loves}]\!]$ $[\![\text{his dog}]\!]$))

(2.13b) $(\bullet)$ (LIFT $[\![\text{his mother}]\!]$) (($\bullet$) ($\bullet$) (($\bullet$) $[\![\text{loves}]\!]$ $[\![\text{his dog}]\!]$))

(2.13c) LIFT $[\![\text{his mother}]\!]$ $\bullet$ (($\bullet$) $\bullet$ ($[\![\text{loves}]\!]$ $\bullet$ $[\![\text{his dog}]\!]$))

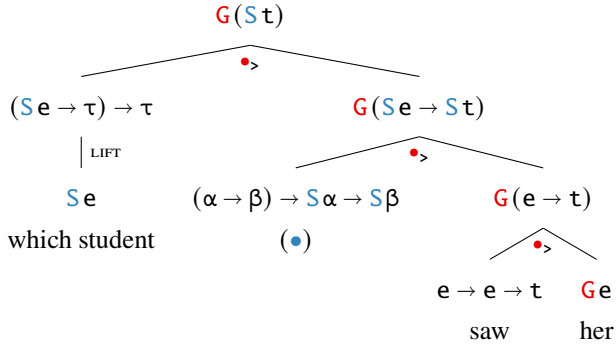
Because all of the effects we've introduced are functorial, and the only interesting aspects of the derivation in (2.14) are the (•)'s, the tree is a template for composition with any of the enriched meanings of Table 2. For instance, switching G for S, derives a multiple-'wh' question as in (2.15).

(2.15)



Moreover, the various instances of (•) are completely independent. All that matters is that the effects of the subject and object are both functorial; but they needn't be *the same* functor. So the template works just as well for sentences containing different kinds of effects, rather than multiple instances of an effect, as in (2.16).

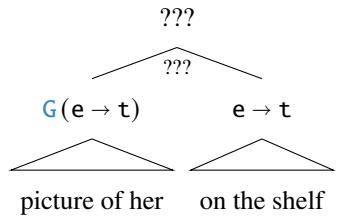
(2.16)



As straightforward as this solution is, there are a few potential reasons for discontent. On the empirical side, effects can pop up just about anywhere, including the daughters of constituents that would otherwise combine via arbitrary modes of combination. As a result, (•) is not the only mode of combination that will need to be mapped. For instance, as things stand there is

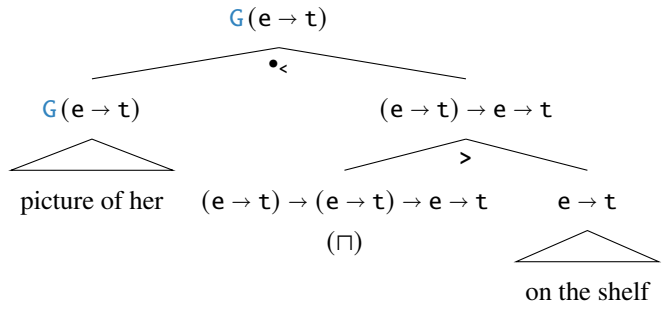
no way to combine an ordinary property with an computational one.

(2.17)



Following (2.14), we would have to treat the Predicate Modification combinator as a lexical item, or at least as a unary type-shifter like LIFT so that it can be partially applied. Doing this would make possible the derivation in (2.18).

(2.18)



In this manner, eventually all modes of combination will need to be realized lexically. Whether this is syntactically justifiable is open to debate, but it certainly increases the distance between the forms that are uttered and the forms that are interpreted. And as a practical matter, the resulting combinatorial system is admittedly unwieldy. Even with practice, derivations are hard to find. Sentences with multiple effects often require a great deal of creativity to compose, mapping and lifting partially applied combinators over constituents.

Anyone who needs convincing of this should try deriving the sentence in (2.14) so that the subject's antecedent-request outscopes the object's. That is, the blue G should precede the red G in the final type signature G(G t). Worse, given that combinators can apply to one another iteratively and without bound, it can be exceedingly difficult to rule analyses out. You never know when some cleverer insertion of maps and lifts would do the trick.

2.2.2 Mapping as a higher-order mode of combination

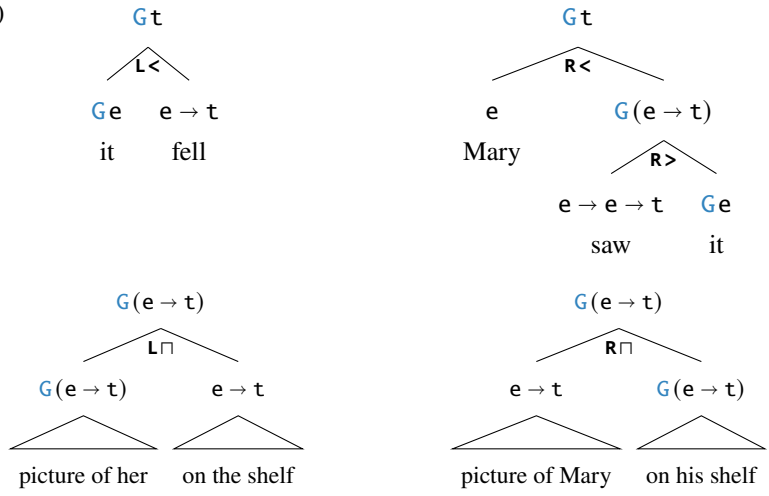
For these reasons, we offer an alternative to the P. Jacobsonian vision. Once more, the key to putting together sentences with multiple non-interacting effects is the ability to map ($\bullet$) itself. But as seen in (2.18), this is not enough. We will want to be able to map an arbitrary mode of combination ($*$) over one of the daughters. In the previous section, this was accomplished by embedding combinators in the object language and using $\bullet_>$ as a binary mode of combination to partially apply them to the relevant daughters one at a time.

Instead, we might just as well provision the grammar with a means of constructing complex combinators from simpler ones, just as complex types are constructed from smaller types. In general, if there is a mode ($*$) that can combine constituents $E_1 :: \sigma$ and $E_2 :: \tau$, then there should also be a mode to combine constituents $E_1 :: \sigma$ and $E_2' :: \Theta\tau$, provided that Θ is a functor. Intuitively, there is a τ thing sitting inside E_2, just waiting to be combined via ($*$) with E_1. So that enriched mode should map ($\lambda b.\, E_1 * b$) over E_2. And likewise if E_1' is of type $\Theta\sigma$. The enriched mode should map ($\lambda a.\, a * E_2$) over E_1'. These **mode-transforming** operations are defined in (2.19). And the complete grammar incorporating these is given in Figure 4.

(2.19) $\quad \mathbf{L}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, a * E_2) \bullet E_1$
$\quad\quad\;\; \mathbf{R}\,(*)\,E_1\,E_2 \coloneqq (\lambda b.\, E_1 * b) \bullet E_2$

With the ability to map arbitrary modes of combination over computations on either side, we can put together syntactically uncluttered derivations with a single effect, no matter where the effect occurs.

(2.20)



Types:

$\tau ::= \mathtt{e} \mid \mathtt{t} \mid \ldots$ Primitive types

$\mid \tau \to \tau$ Function types

$\mid \tau \times \tau$ Product types

$\mid \tau + \tau$ Sum types

$\mid \{\tau\}$ Powerset types

$\mid \Sigma\tau$ Computation types

Effects:

$\Sigma ::= \mathtt{G}$ Reading

$\mid \mathtt{W}$ Writing

$\mid \mathtt{S}$ Indeterminacy

$\mid \ldots$

Basic Combinators:

$(\mathbf{>}) :: (\alpha \to \beta) \to \alpha \to \beta$ Forward Application

$f \mathbf{>} x \coloneqq f\,x$

$(\mathbf{<}) :: \alpha \to (\alpha \to \beta) \to \beta$ Backward Application

$x \mathbf{<} f \coloneqq f\,x$

$(\sqcap) :: (\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{e} \to \mathtt{t}$ Predicate Modification

$f \sqcap g \coloneqq \lambda x.\, f\,x \wedge g\,x$

$\ldots$

Meta-combinators:

$\mathbf{L} :: (\sigma \to \tau \to \omega) \to \Theta\sigma \to \tau \to \Theta\omega$ Map Left

$\mathbf{L}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, a * E_2) \bullet E_1$

$\mathbf{R} :: (\sigma \to \tau \to \omega) \to \sigma \to \Theta\tau \to \Theta\omega$ Map Right

$\mathbf{R}\,(*)\,E_1\,E_2 \coloneqq (\lambda b.\, E_1 * b) \bullet E_2$

Figure 4 A type-driven grammar with functorial effects

2.3 Effect layering

Importantly, the operators in (2.19) are iterative in the sense that they take a binary mode $(*)$ and return a new binary mode, $\mathbf{L}*$ or $\mathbf{R}*$. This means they can in principle apply to their own output. For instance, since $\mathbf{L}*$ is a binary mode, $\mathbf{L}(\mathbf{L}*)$ is yet another mode, as is $\mathbf{R}(\mathbf{L}*)$, and likewise for $\mathbf{L}(\mathbf{R}*)$ and $\mathbf{R}(\mathbf{R}*)$.

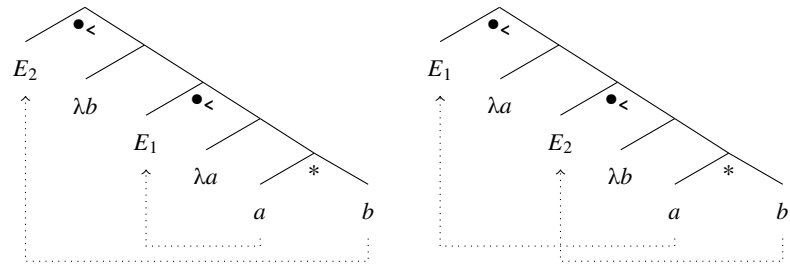
What do these higher-order combinators amount to? Particularly illuminating are the cases where both daughters are computations: $\mathbf{L}(\mathbf{R}*)$ and $\mathbf{R}(\mathbf{L}*)$. Cranking through the definitions gives:

(2.21) $\mathbf{L}(\mathbf{R}*) = \lambda E_1 \lambda E_2.\ (\lambda a.\ (\lambda b.\ a * b) \bullet E_2) \bullet E_1$
$\mathbf{R}(\mathbf{L}*) = \lambda E_1 \lambda E_2.\ (\lambda b.\ (\lambda a.\ a * b) \bullet E_1) \bullet E_2$

Just for illustrative purposes, let us rewrite these equations with the order of $(\bullet)$'s arguments flipped, so that the computation comes first and the to-be-mapped function second. That is, let's swap out $(\bullet)$ for $(\bullet_<)$, making the relevant adjustments. This gives the equations in (2.22).

(2.22) $\mathbf{L}(\mathbf{R}*) = \lambda E_1 \lambda E_2.\ E_1 \bullet_< (\lambda a.\ E_2 \bullet_< (\lambda b.\ a * b))$
$\mathbf{R}(\mathbf{L}*) = \lambda E_1 \lambda E_2.\ E_2 \bullet_< (\lambda b.\ E_1 \bullet_< (\lambda a.\ a * b))$

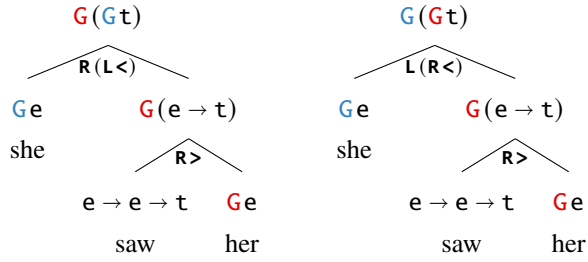
In this form, the derived higher-order modes reveal a striking resemblance to derivations invoking **Quantifier Raising** (though most functors have nothing to do with quantification and there is certainly no raising), as illustrated below.



We discuss the relationship between scope and effects in Chapter 4, but it is worth noting that the diagrams above already suggest a sense in which computation-denoting constituents **take scope** over their compositional contexts. Choosing to map anything over the $\mathbf{L}$-eft daughter gives the left daughter's effect priority over whatever is mapped. For instance, if both daughters request antecedents, then the left daughter's request will come first. Choosing instead to

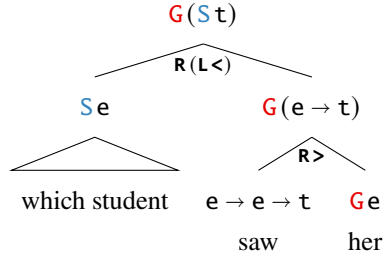
map something over the **R**-ight daughter gives the right daughter's effect priority. The difference can be seen in (2.23).

(2.23)

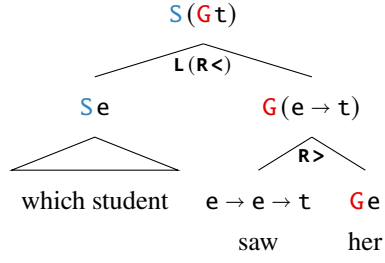


What this priority amounts to depends on the nature of the effect. But because the operations involved here work for any functorial constructor, we can immediately combine constituents with different kinds of effects, often in multiple ways. For instance, a context-sensitivite predicate and an indeterminate subject can be combined in the two ways shown in (2.24).

(2.24a)



(2.24b)



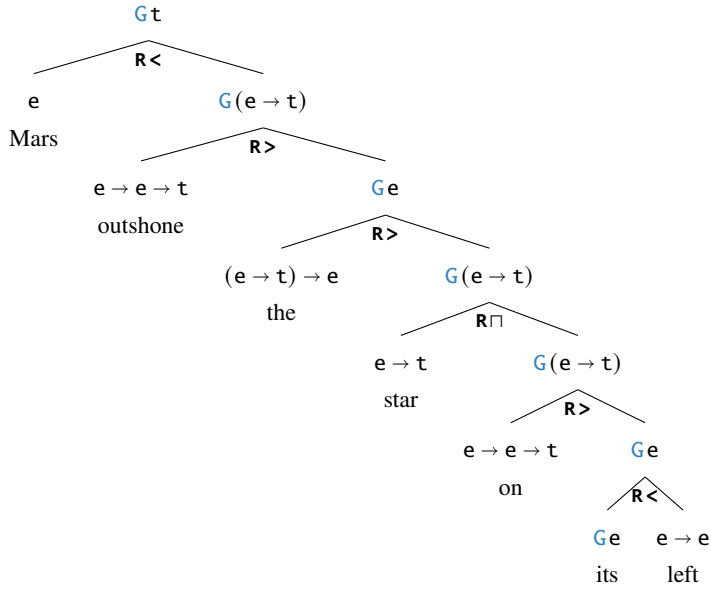
As it happens, both of these denotations have been proposed in the literature on questions. The former, (2.24a), appears in Charlow (2020); Hagstrom (1998); Hamblin (1973); Kratzer and Shimoyama (2002), among others. The latter,

(2.24b), appears in, e.g., Poesio 1996; Romero and Novel 2013; Rooth 1985. Recognizing that both of these effects are functorial and allowing a more flexible approach to composition means that for sentences like this, there is no reason to choose one over the other. We needn't generalize to either potential "worst case".

2.4 Functors and pseudoscope

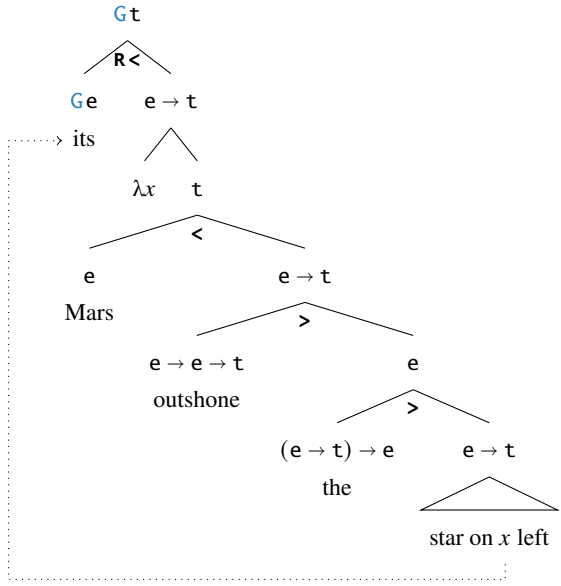
Absent any sort of closure operators, which we discuss in Chapter 3, functorial effects **percolate** up the tree in which they're composed. This is plainly evident from the types. In (2.25), for example, the genitive pronoun embedded in the object introduces an anaphoric dependency to the denotation, and that dependency is inherited by every node dominating it in the tree.

(2.25)



Once composed, the entire sentence becomes context-dependent; it requests an antecedent in order to compute a truth value. In a sense, then, the pronoun's computational effect — requesting an antecedent — is displaced from the location of the pronoun itself. In fact, the whole derivation is equivalent to what we'd get if we "Quantifier"-Raised the pronoun and mapped in its syntactic context, as in (2.26).

(2.26)



With a little effort, this equivalence can be seen to follow from the Law of Composition (2.6). Just rewriting the law as in (2.27) using the combinators introduced in this chapter, together with a suggestive arrow, gets most of the way there.

(2.27) $f \bullet_{>} (g \bullet_{>} M) = M \overset{\frown}{\bullet}_{<} (\lambda x.\, f > (g > x))$

Since the equivalence is algebraic rather than due to some quirk of context-dependence, it guarantees that for *any* functorial effect, mapping can be seen as a means of giving the effect scope over its compositional context. That is, repeatedly mapping over an embedded computation is equivalent to scoping the computation out of the way and mapping once where it lands.

But importantly, the scope provided by $(\bullet)$ does not depend on any assumptions about syntactic transformations. In particular, there is no reason to expect effect-percolation to exhibit sensitivity to the sorts of **islands** that govern movement. And indeed, all of the effects in Table 2 — with the exception of quantifiers, which we return to in Chapter 5 — are island *in*sensitive.

For instance, consider the examples in (2.28).

(2.28a) Who remembers when who left?

(2.28b) Mary only gets mad when JOHN leaves the lights on

(2.28c) Mary said that if John's car is in the garage, then he's home

(2.28d) Mary knows that if her cat, named Sassy, is home, then John is too

These sentences all contain an effect-denoting expression within an island, either an embedded question or embedded adjunct. Yet they all also have readings in which the semantic force of that embedded expression is felt outside of the island. For example, the question in (2.28a) has a reading in which it asks which pairs of people $\langle a, b \rangle$ are such that a knows when b left. The sentence in (2.28b) declares John to be the only person such that Mary gets mad when he leaves the lights on. (2.28c), as a whole, presupposes that John has a car, even though the presupposition trigger 'John's car' is in an embedded hypothetical. Likewise, (2.28d) commits the speaker to Mary's cat being named Sassy, regardless of what Mary knows or whether her cat is home.

In these construals, the effect-generating expressions are sometimes said to take **exceptional scope** over the islands that embed them. The capacity for exceptional scope is a hallmark of a functorial effect.

2.5 Implementing functorial effects in the type-driven interpreter

Notice that, as in the single-effect derivations of (2.20), the new multi-effect derivations are syntactically spare. Nothing is inserted and no types are shifted. In other words, composition remains type-driven in the sense that every possible way of interpreting a combination of two expressions is determined by their types.

As discussed in Chapter 1, one of the main benefits of this approach is that it takes the creativity out of composition. Given the grammar in Figure 4, there is an effective procedure for determining all the possible combinations of any two types, just as there was for the basic grammar in Figure 1. In this section, we extend the interpreter of Section 1.5 to cover the grammar of Figure 4.

First, we need to expand our representation of types to incorporate type constructors modeling effects. Following Figure 4, we say that a type `Ty` can be atomic or functional, as before, but also now computational. A computation type is parameterized by an effect `Eff`, which we discuss below.

```
data Ty
  = E | T          -- primitive types
  | Ty :-> Ty      -- function types
  | Comp Eff Ty    -- computation types
```

Next, we expand our inventory of combinatory modes. The new modes **R** and **L** are meta-combinators; they take modes as arguments and return modes.

(2.29) $$\mathbf{R}\,(*)\,E_1\,E_2 \coloneqq (\lambda b.\, E_1 * b) \bullet E_2$$
$$\mathbf{L}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, a * E_2) \bullet E_1$$

Our representations `MR` and `ML` meta-combinators are thus parameterized by this underlying mode $(*)$ :: `Mode`.

```
data Mode
  = FA | BA | PM       -- basic modes of combination
  | MR Mode | ML Mode  -- map right and map left
```

As far as type-driven combination is concerned, the only thing we need to know about an effect is its label, and whether or not it is a functor. None of the grammatical, combinatoric operations inspect the internal structure of an effect. Indeed, this is the whole point of the algebraic abstractions. Knowing that an effect `θ` is functorial is enough to know that it can be combined using **R**/**L**. Of course the actual semantics will depend on how the effect is encoded (is it a product, a set, a function into sets, etc.?) and how $(\bullet)$ is defined for that encoding, but the type logic itself needn't bother with such matters.

Consequently, the representation of effects `Eff` includes just enough information to drive the combinatorics, namely a label indicating what kind of effect it is and parameters for whatever incidental data the computation is specialized to (the type of the environment it reads from, or the type of the data that it stores, or the type of context it quantify overs, etc.). Note that all of the effects we consider here are functors, so the `functor` predicate happens to be vacuous. But we include it for good measure, and to set the stage for future chapters.

```
data EffX
  = SX        -- computations with indeterminate results
  | GX Ty     -- computations that query an environment of type Ty
  | WX Ty     -- computations that store information of type Ty
  | CX Ty Ty  -- computations that quantify over Ty contexts
  -- and so on for other effects, as desired

functor :: EffX -> Bool
functor _ = True
```

With these representations fixed, we define the logic of combination. This is again a simple matter of pattern-matching on the types of the daughters. For starters, if the daughters `l` and `r` can be combined via any of the basic modes of combination from Chapter 1, then go for it. Recall that the function `modes` returns a list containing whatever basic `Mode`s are applicable to combining `l` and `r`, together with the `Ty`pe that would result from so combining them.

In addition, we check for two other possibilities. The function `addMR` returns an empty list — adding no new modes of combination to what the basic `modes` was able to find — unless the right daughter's type is a computation type `Comp f t` with a functorial effect `f`. If it is, then we try to `combine` the left daughter `l` with the right daughter's underlying type `t`. That recursive call will produce a list of possible `Mode`s and resulting `Ty`pes. If there is no way to combine `l` and `t`, then the new list will again be empty, adding nothing to the basic `modes`. But if it is possible the combine `l` and `t`, then for each way of doing so `(op, u)`, we build a new higher-order mode `MR op`, signaling that `op` can be mapped over the right daughter, resulting in a combined type `Comp f u`.

The case for checking that the left daughter `l` is functorial is exactly symmetric to the right one. Importantly, these two investigations `addMR` and `addML` are not exclusive. If both daughters are functorial, and the underlying types can be combined, they will both return new substantive modes of combination.

```
combine :: Ty -> Ty -> [(Mode, Ty)]
combine l r =
  -- see if any basic modes of combination work
  modes l r
  -- if the right daughter is functorial, try to map over it
  ++ addMR l r
  -- if the left daughter is functorial, try to map over it
  ++ addML l r

addMR l r = case r of
  Comp f t | functor f
    -> [ (MR op, Comp f u) | (op, u) <- combine l t ]
  _ -> [                                             ]

addML l r = case l of
  Comp f s | functor f
    -> [ (ML op, Comp f u) | (op, u) <- combine s r ]
  _ -> [                                             ]
```

Finally, the top-level interpreter that annotates trees is exactly as it was in Chapter 1, except that the basic `modes` function is upgraded to the recursive `combine` function.

```
synsem :: Syn -> [Sem]
synsem syn = case syn of
  (Leaf t w)          -> [Lex t w]
  (Branch lsyn rsyn) ->
    [ Comb ty op lsem rsem
      | lsem     <- synsem lsyn
      , rsem     <- synsem rsyn
      , (op, ty) <- combine (getType lsem) (getType rsem) ]
  where
    getType (Comb ty _ _ _) = ty
```

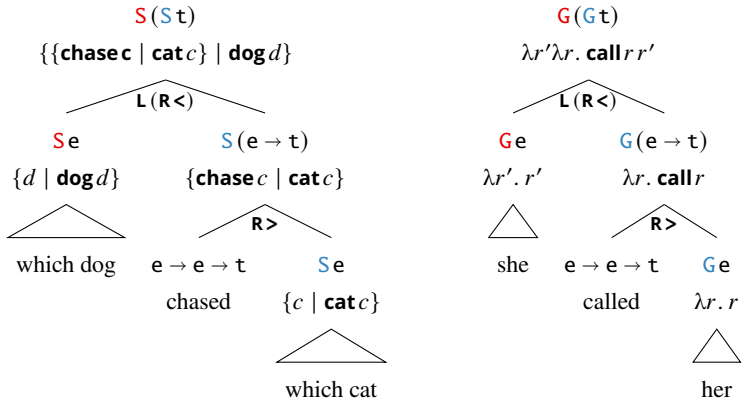
3 Applicative Functors

3.1 Effects that amalgamate

As seen in Chapter 2, one of the benefits of using (•) or **R** / **L** for composition is that it accommodates any number and variety of computations. One way or another, the computational side effects, whatever their shape, are passed over in order to get at the underlying argument-structural results and put them together.

However, this means that when two effects are combined, the result is necessarily **higher-order**, a computation that returns another computation. For instance, with two 'wh'-expressions, we end up with a set of sets of propositions. With two pronouns, we end up with a function from an antecedent to a function from an antecedent to a proposition. And so on.

(3.1)



Even among theories that countenance these sorts of higher-order meanings, they are not generally thought to represent the default interpretations of such sentences, much less their only interpretations. Consider, for instance, the typical **variable-full** format for managing pronouns (Heim & Kratzer, 1998). Every constituent is evaluated relative to a variable assignment, whether it contains pronouns or not. If it does, those pronouns project particular coordinates of the assignment. If it does not, the assignment is ignored. And importantly, when two sisters are both evaluated, they are evaluated at *the same* assignment.

A fragment with this shape is outlined in Figure 5. Compared to the grammar in Figure 4, several things stand out. The environment-sensitivity effect is always outermost in a type; every expression's type is of the form $\mathtt{G}\sigma$, for some ordinary type σ. Accordingly, all of the modes of combination expect their daughters to be

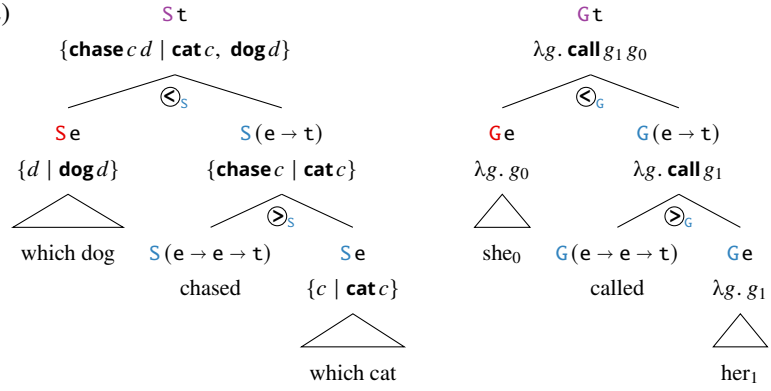
environment-sensitive, and return an environment-sensitive result. This means that lexical items themselves must all be coerced into environment-reading computations, whether they pay any attention to the environment or not.

Exactly the same pattern emerges in a standard Hamblin grammar for questions (Hamblin, 1973), as sketched in Figure 6. Every constituent denotes an indeterminate computation — modeled by the set of values it might return — whether that constituent contains a 'wh'-expression or not. If it does, then the 'wh'-expressions generate genuine alternatives. If it does not, then the denotation is a singleton value. Importantly, when two sisters are both evaluated, they are evaluated *pointwise*, so that the alternatives generated in the two daughters are amalgamated into a single large set.

Again, as seen in Figure 6, indeterminacy is pervasive and top-level; every expression's type is of the form $\mathsf{S}\,\sigma$. All the modes of combination expect indeterminate daughters, and return indeterminate results. And all lexical items are coerced into indeterminate computations, whether they generate any alternatives or not.

Putting these grammars to work is straightforward, at least in examples like those in (3.1).

(3.2)



However, both of these grammars are examples of generalization to the worst case, forcing every constituent to be as structurally complex as the constituents of semantic interest. Unfortunately, neither case is as bad as it can get. Simply mixing the two kinds of phenomena is beyond the reach of either grammar. The rigid, pervasive replacement of ordinary combinatory modes with effect-specific variants limits the applicability of the fragment to just the specific effects described. What is gained in uniformity and simplicity is sacrificed in flexibility and extsensibility.

Types:

$\sigma ::= \mathsf{e} \mid \mathsf{t} \mid \ldots$ Primitive pre-types

$\mid \sigma \to \sigma$ Function pre-types

$\tau ::= \mathsf{G}\sigma$ Expression types

Combinators:

$(\gg_{\mathsf{G}}) :: \mathsf{G}(\alpha \to \beta) \to \mathsf{G}\alpha \to \mathsf{G}\beta$

$F \gg_{\mathsf{G}} X := \lambda g.\, F\, g\, (X\, g)$

$(\ll_{\mathsf{G}}) :: \mathsf{G}\alpha \to \mathsf{G}(\alpha \to \beta) \to \mathsf{G}\beta$

$X \ll_{\mathsf{G}} F := \lambda g.\, F\, g\, (X\, g)$

...

Lexicon:

$\text{it}_n :: \mathsf{G}\,\mathsf{e}$

$:= \lambda g.\, g_n$

Mars $:: \mathsf{G}\,\mathsf{e}$

$:= \lambda g.\, \mathbf{m}$

cat $:: \mathsf{G}(\mathsf{e} \to \mathsf{t})$

$:= \lambda g.\, \mathbf{cat}$

...

Figure 5 Env.-sensitive grammar

Types:

$\sigma ::= \mathsf{e} \mid \mathsf{t} \mid \ldots$ Primitive pre-types

$\mid \sigma \to \sigma$ Function pre-types

$\tau ::= \mathsf{S}\sigma$ Expession types

Combinators:

$(\gg_{\mathsf{S}}) :: \mathsf{S}(\alpha \to \beta) \to \mathsf{S}\alpha \to \mathsf{S}\beta$

$F \gg_{\mathsf{S}} X := \{f\, x \mid f \in F,\ x \in X\}$

$(\ll_{\mathsf{S}}) :: \mathsf{S}\alpha \to \mathsf{S}(\alpha \to \beta) \to \mathsf{S}\beta$

$X \ll_{\mathsf{S}} F := \{f\, x \mid f \in F,\ x \in X\}$

...

Lexicon:

who $:: \mathsf{S}\,\mathsf{e}$

$:= \{x \mid \mathbf{person}\, x\}$

Mars $:: \mathsf{S}\,\mathsf{e}$

$:= \{\mathbf{m}\}$

cat $:: \mathsf{S}(\mathsf{e} \to \mathsf{t})$

$:= \{\mathbf{cat}\}$

...

Figure 6 Indeterminate grammar

3.2 Applicative Functors

Fortunately, the underlying strategy of both the Heim and Kratzer (1998) grammar for environment-sensitivity and the Hamblin (1973) grammar for interrogativity can be made modular and algebraic, in line with the generic mapping operation ($\bullet$) of Chapter 2. The essential components of both strategies is a pair of operations: foremost, a means of composing a computation that yields a function $\Theta(\alpha \to \beta)$ with one that yields an argument $\Theta\alpha$ to form a computation yielding a result $\Theta\beta$; and secondarily, a means of injecting an ordinary value α into a **unitary**, or "trivial", computation.

Intuitively, a computation is trivial if it adds no effect of any consequence. It is a computation that does nothing except return a value. A trivial environment-sensitive computation is one that requests an environment but makes no use of it, so that it doesn't *really* read from the environment at all. A trivial indeterminate computation is one that computes exactly one thread, so that there isn't *really* any parallelism to speak of.

Technically, what it means for a computation to be trivial depends on how effect-generating functions and effect-generating arguments are combined, and in particular on how the effects that they generate are combined. This is the only way to know whether the potentially trivial effect really does not change anything. One natural abstraction for this sort of relationship is known as an **applicative functor** (Kiselyov, 2015; McBride & Paterson, 2008), which we'll call an applicative for short.

A type constructor Θ is an applicative functor if there are operations η and $(\circledast)$ with the types indicated in (3.3) respecting the laws in (3.4).

(3.3)
$$\eta :: \alpha \to \Theta\alpha$$
$$(\circledast) :: \Theta(\alpha \to \beta) \to \Theta\alpha \to \Theta\beta$$

(3.4)

Homomorphism
$\eta f \circledast \eta x = \eta(f x)$

Identity
$\eta(\lambda x.\, x) \circledast v = v$

Interchange
$\eta(\lambda f.\, f x) \circledast u = u \circledast \eta x$

Composition
$\eta(\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$

In Haskell, these operations are known as `pure` and `ap`.

```
class Functor f => Applicative f where
  pure :: a -> f a
  ap :: f (a -> b) -> f a -> f b
```

It's easy to see that the combinators and implicit lexical coercion mechanisms of Figures 5 and 6 constitute G and S applicative functors, respectively:

(3.5)
$$\eta_{\mathsf{G}}\, x \coloneqq \lambda g.\, x \qquad\qquad \eta_{\mathsf{S}}\, x \coloneqq \{x\}$$
$$F \circledast_{\mathsf{G}} X \coloneqq \lambda g.\, F\, g\, (X\, g) \qquad F \circledast_{\mathsf{S}} X \coloneqq \{f\, x \mid f \in F,\ x \in X\}$$

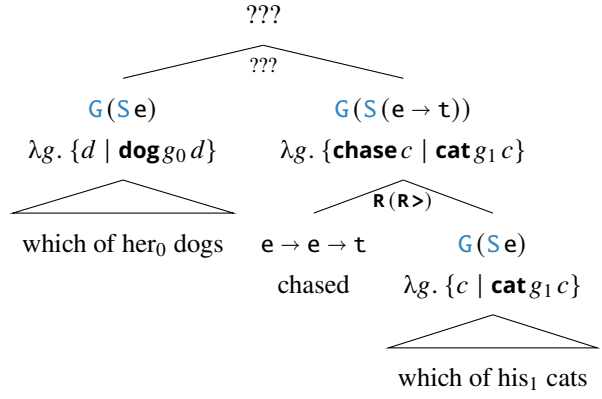
In fact, all of the effects in Table 2 have corresponding applicative instances. We provide the standard applicative instances of these types in Appendix A3. For some of the effects, there are even multiple ways of defining $(\circledast)$ that satisfy the laws. And as might be expected from the terminology, all applicative functors are functors, in that any way of defining η and $(\circledast)$ that satisfy the laws in (3.4) will determine a way of defining $(\bullet)$ that satisfies the laws in (2.6), by way of the equation in (3.6).

(3.6) $k \bullet m \equiv \eta\, k \circledast m$

Putting these operations to work in deriving natural language meanings looks much the same as it did in Chapter 2. Clearly $(\circledast)$ might be added without any imagination as a potential mode of combination. This is just the strategy of the grammars in Figures 5 and 6 above. The coercion operator η is unary, unlike $(\bullet)$ and $(\circledast)$, so could be added as a **type-shifter**, like LIFT. Then derivations would then look much the same as in (3.1), except that the transitive verbs would be shifted by η before combination.

The trouble with this straightforward approach is also the same as in Chapter 2. There is no general way to combine constituents with multiple effects. That is, while $(\circledast)$ guarantees a way to put together two type-S constituents, or two type-G constituents, it does not by itself suffice as a means to combine two GS constituents, or two SG constituents. And such constituents are by no means exotic. All it takes is a left branch with an alternative generator and a pronoun, and a right branch with an alternative generator and a pronoun, as in (3.7).

(3.7)

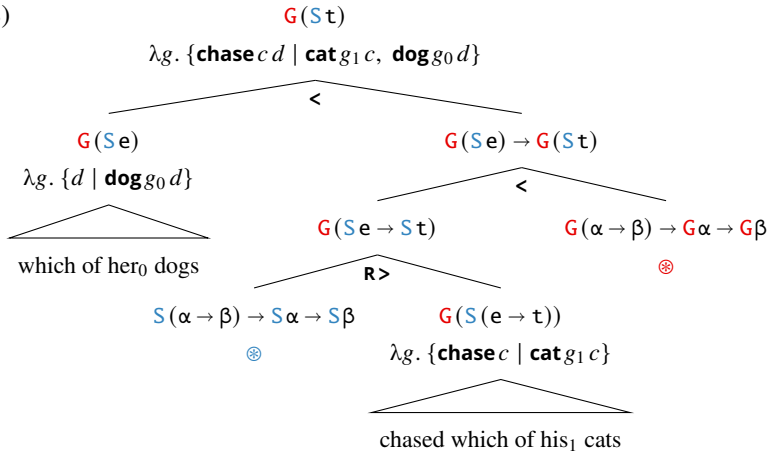


The solutions to higher-order applicative combination are again the same as in Chapter 2. The problem is that $\mathsf{G}(\mathsf{S}\,\mathsf{e})$ and $\mathsf{G}(\mathsf{S}\,(\mathsf{e}\to\mathsf{t}))$ cannot be combined via $\circledless_{\mathsf{G}}$ because neither G-computation delivers a function. They both deliver further computations: $\mathsf{S}\,\mathsf{e}$ and $\mathsf{S}\,(\mathsf{e}\to\mathsf{t})$. But of course, *those* computations can be combined via $\circledless_{\mathsf{S}}$. And since G is an applicative functor, we should be able to map the $\circledless_{\mathsf{S}}$ mode of combination over the outer G layer, merging the two G effects in the process.

3.2.1 Applicatives in the language

Adding η and $(\circledast)$ as polymorphic lexical items certainly opens the door to this sort of higher-order mapping (Charlow, 2018, 2022).

(3.8)



But this approach faces all the same challenges as adding $(\bullet)$ to the object language did in Chapter 2. Namely, it takes a great deal of ingenuity to find derivations, when they exist, and even more ingenuity to figure out when they don't. And the derivations that result may be full of syntactically suspect combinatory operators.

3.2.2 Structured application as a higher-order mode of combination

So we follow the strategy of Section 2.2.2, provisioning the grammar with a means of applying an arbitrary combinator to the underlying values of two computations, amalgamating their effects in a singly-layered structure. That is, whenever there is a mode $(*)$ that can combine constituents $E_1 :: \sigma$ and $E_2 :: \tau$, then there should also be a mode to combine constituents $E_1' :: \Theta\,\sigma$ and $E_2' :: \Theta\,\tau$, if Θ is an applicative functor. Intuitively, there is a σ thing sitting inside E_1', and a τ thing sitting inside E_2', both ready to be combined via $(*)$. And since Θ is applicative, there is a way of zipping the computational contexts of E_1' and E_2' together. The relevant higher-order combinator is defined in (3.9).

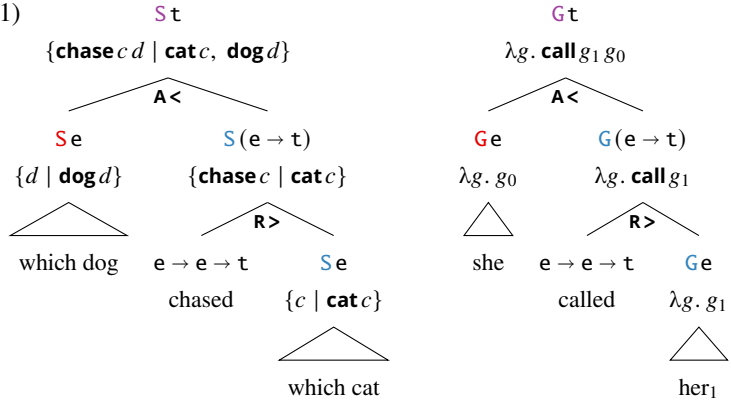
(3.9) $\quad \mathbf{A}\,(*)\,E_1\,E_2 \coloneqq \eta\,(*) \circledast E_1 \circledast E_2$

Then the modes of combination from the Heim and Kratzer and Hamblin grammars in Figures 5 and 6 are recovered as in (3.10).

(3.10) $\quad \text{⊱} \equiv \mathbf{A}\,(>)$
$\quad\quad\;\; \text{⊰} \equiv \mathbf{A}\,(<)$

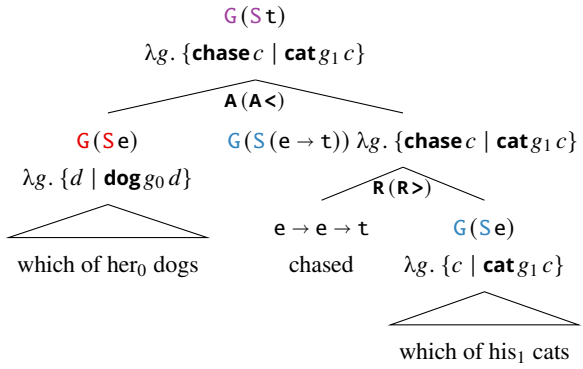
And the earlier derivations involving one kind of effect at a time are recovered as in (3.11).

(3.11)



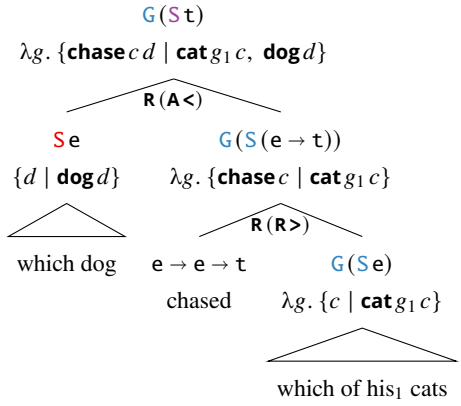
The troubling composite-effect sentence of (3.7) is now derived as in (3.12).

(3.12)

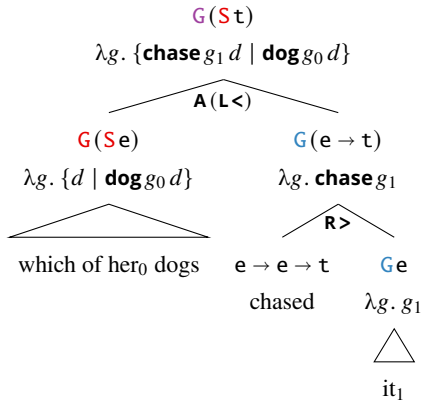


The combination of **R/L** and **A** provides a lot of flexibility. For instance, the effects needn't be balanced the way they are in (3.12) in order to take advantage of (⊛). Examples of computationally imbalanced daughters are shown in (3.13) and (3.14). Outer effects may be amalgamated with **A** while inner ones are mapped over, or vice versa.

(3.13)



(3.14)



3.3 Selectivity and unselectivity

As discussed in Section 2.4, the functoriality of an effect gives rise to a kind of exceptional scope-taking. Barriers to movement obviously exert no direct influence on the scope of an effect, since the generators of effects do not move (or certainly do not need to move for their effect to spread upward). And barriers to quantifier scope, if they are separate from those for movement, likewise hold no particular sway over the percolation of effects since effects are not generally quantificational. In other words, island boundaries are simply mapped over effectful computations, just like ordinary predicates.

But this is not to say it is impossible to operate on a computation itself, as opposed to the value it computes. Such operators, which we will refer to broadly as **closure operators**, come in two flavors, distinguished more by their linguistic role than their formal properties. On the one hand, there are linguistic items thought to **associate with an effect** in an essential capacity. The way we will use the term, an expression associates with an effect θ if it takes an argument of type $\theta\,\alpha$, for some type α. That is, the expression's semantics is *expecting* a particular type of computation. The canonical examples in this category are **focus-sensitive items** like 'only' and 'also' (Rooth, 1985).

(3.15) $\text{only} :: \mathtt{F}\,\mathtt{t} \to \mathtt{t}$

$[\![\text{only}]\!] \coloneqq \lambda m.\, \{p \in \mathbf{snd}\, m \mid p\} = \{\mathbf{fst}\, m\}$

We would also include alternative-sensitive expressions in this category like Japanese 'mo' and 'ka' (Kratzer & Shimoyama, 2002; Shimoyama, 2006) or even ordinary question-embedding attitudes like 'wonder' (Groenendijk & Stokhof, 1984). Likewise, any dynamic semantic operation, such as is often proposed for negation or quantificational determiners (e.g., Muskens (1990, 1996)), would count as associating with the dynamic effects generated by their arguments. The same goes for operators that bind pronouns, altering the environments that their prejacents are evaluated in. And so on.

(3.16a) $\text{mo} :: \mathtt{S}\,\mathtt{t} \to \mathtt{t}$

$[\![\text{mo}]\!] \coloneqq \lambda m.\, \bigwedge m$

(3.16b) $\text{wonder} :: \mathtt{S}\,\mathtt{t} \to \mathtt{e} \to \mathtt{t}$

$[\![\text{wonder}]\!] \coloneqq \lambda m \lambda x.\, \mathbf{wonder}\, m\, x$

(3.16c) $\text{not} :: \mathtt{D}\,\mathtt{t} \to \mathtt{D}\,\mathtt{t}$

$[\![\text{not}]\!] \coloneqq \lambda m \lambda s.\ \{\langle \neg \exists x.\ \langle x,\ \mathbf{T}\rangle \in m\, s,\ \ s\rangle\}$

(3.16d) $\lambda_n :: \mathtt{G}\beta \to \mathtt{e} \to \mathtt{G}\beta$

$[\![\lambda_n]\!] \coloneqq \lambda m \lambda x \lambda g.\ m\, g[n \mapsto x]$

On the other hand, there are a smattering of covert operators usually associated with some sort of complete **evaluation domain**, often a clause. Semantically, these sorts of operators might be thought of as *executing* the computations that their prejacents denote, and evaluating the results. The most well-known examples are the many varieties of **existential closure** used in alternative- and dynamic-semantic settings (e.g., Heim 1982; Kratzer and Shimoyama 2002). But also possibly in this category are various **exhaustivity** operators (e.g., Krifka 1995), the "lowering" operations of continuation semantics (Barker, 2002; Barker & Shan, 2014), and any mechanism for locally accommodating a presupposition (e.g., Beaver and Krahmer 2001).

(3.17a) $\exists\text{-clo} :: \mathtt{S}\,\mathtt{t} \to \mathtt{t}$

$[\![\exists\text{-clo}]\!] \coloneqq \lambda m.\ \bigvee m$

(3.17b) $\text{lower} :: \mathtt{C}\,\mathtt{t} \to \mathtt{t}$

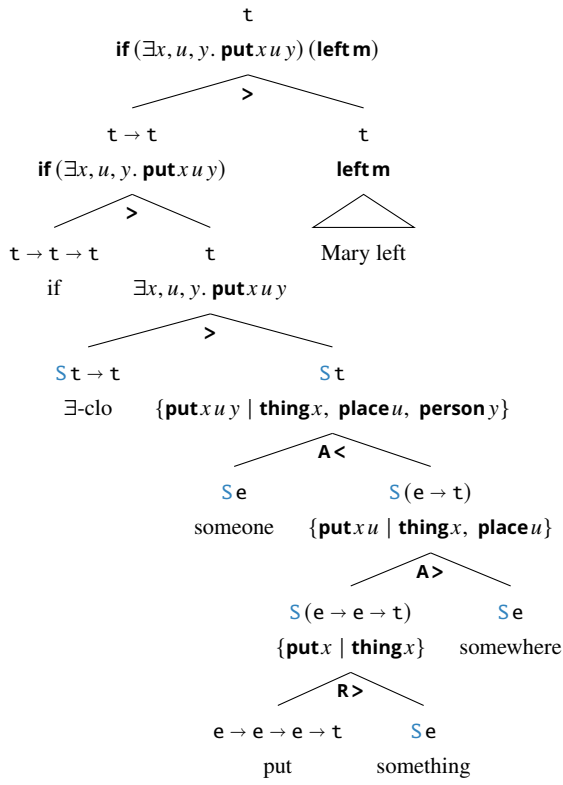
$[\![\text{lower}]\!] \coloneqq \lambda m.\ m\,(\lambda p.\ p)$

(3.17c) $\text{Assert} :: \mathtt{M}\,\mathtt{t} \to \mathtt{t}$

$[\![\text{Assert}]\!] \coloneqq \lambda m.\ \mathbf{F}$ if $m = \#$ else m

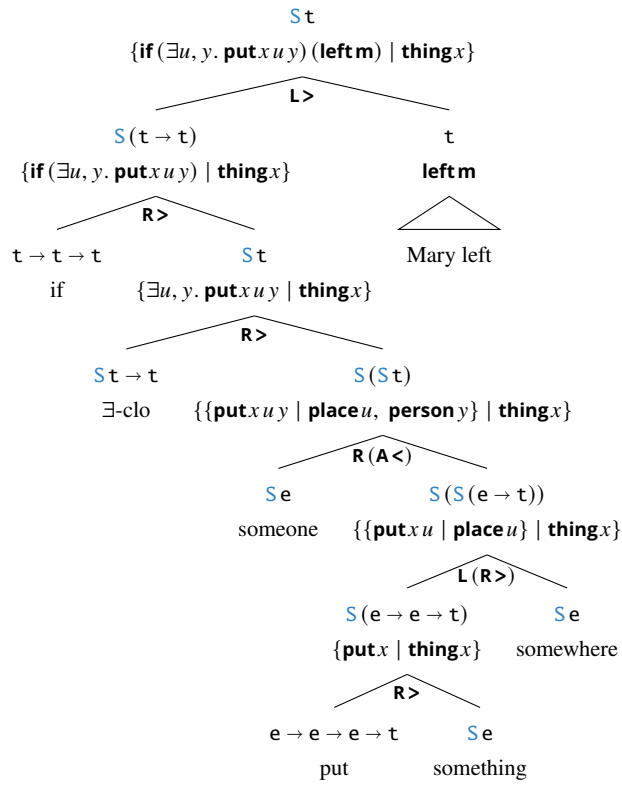
When a constructor $\mathtt{\Theta}$ is applicative, any operator $\lightning :: \mathtt{\Theta}\,\sigma \to \tau$ will in principle close over every effect in its scope simultaneously. Operators like this are sometimes said to be **unselective**, on analogy with the unselective binding of indefinites found in Lewis (1975) and Heim (1982). For instance, consider the conditional in (3.18). Thanks to the applicativity of S, a single existential closure operator manages to capture the scope of all the indefinites in the antecedent.

(3.18)



At the same time, every applicative functor is still a functor, which suffices to establish higher-order derivations of the antecedent. Unless something precludes such derivations, this predicts the availability of various exceptional-scope readings. For instance, (3.19) mimics the derivation in (3.18) except that the alternatives generated by the direct object are consistently mapped-over at every node. The alternatives generated by the other two indefinites are merged as above and jointly closed over in the antecedent.

(3.19)



Even a single effect may in principle escape the scope of a closure operator, with the help of η. For demonstrative purposes, imagine the closure operator of (3.19) is obligatory, or perhaps even part of the semantics of the conditional.[2] Then the only way to combine a conditional with an ordinary, effect-free

[2]Conditionals are indeed occasionally thought to associate with static alternatives of the S variety (Alonso-Ovalle, 2009), as in (3.20), and almost always thought to associate with dynamic alternatives of the D variety (e.g., Muskens 1996), as in (3.21).

(3.20) $\text{if} :: \mathtt{S}\,\mathtt{t} \to \mathtt{S}\,\mathtt{t} \to \mathtt{t}$

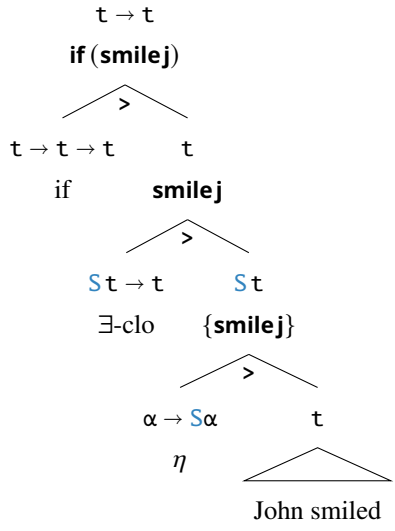
$[\![\text{if}]\!] \coloneqq \lambda m \lambda n.\ \bigwedge\{p \Rightarrow \bigvee n \mid p \in m\}$

(3.21) $\text{if} :: \mathtt{D}\,\mathtt{t} \to \mathtt{D}\,\mathtt{t} \to \mathtt{D}\,\mathtt{t}$

$[\![\text{if}]\!] \coloneqq \lambda m \lambda n \lambda i.\ \{\langle \forall j.\ \langle \mathbf{T},\ j\rangle \in m\, i \Rightarrow \exists k.\ \langle \mathbf{T},\ k\rangle \in n\, j,\ \ i\rangle\}$

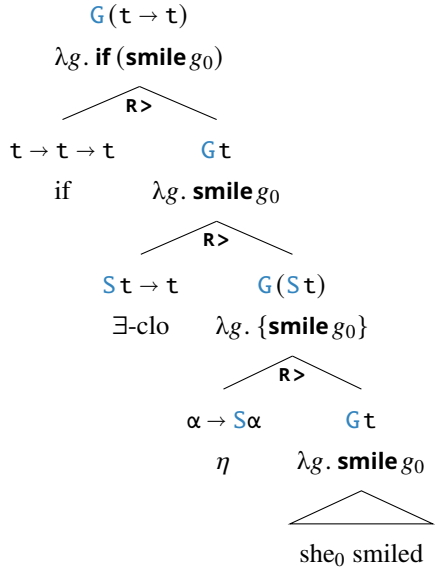
antecedent would be to coerce the antecedent into a trivially indeterminate computation, one whose only thread is the ordinary proposition.

(3.22)



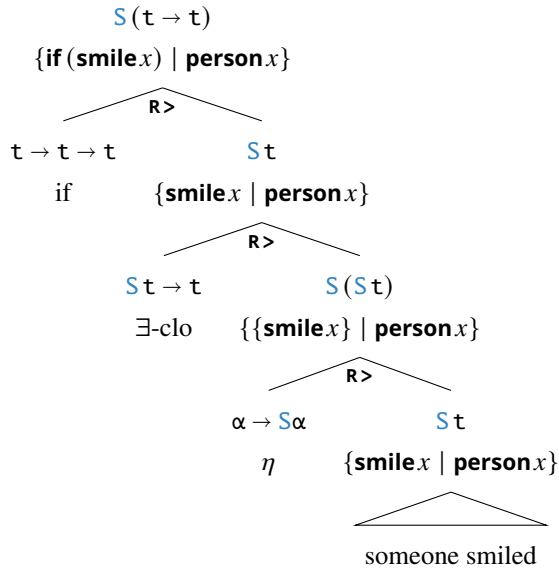
If the antecedent has an effect, but not an effect that ∃-clo knows what to do with, then the η-coercion must apply to the underlying, pure value of the antecedent.

(3.23)



This same strategy, of applying pure to *the result* of a computation — corresponding to its underlying type — might just as well be used to coerce the propositions underlying an S-node, as in (3.24). In so doing, the existential force of the indefinite, carried by the alternatives it generates, escapes the closure operator and outscopes the conditional in which it appears.

(3.24)



In this manner, cleverly allocated η operators can play two apparently contrary roles in the grammar. They can inject pure values into computations that *feed* closure operators, creating the effect structure that the operators need. But in exactly the same way, they can render the underlying values of already-effectful denotations as dummy computations, providing decoy targets for closure operators, and in the process *shield* effects from the operators that threaten to consume them.

However, as in other places in this Element, we should like very much to eliminate cleverness from the picture. Once a theorist has fixed a grammar and a set of lexical items, it should not require an act of inspiration to figure out the range of meanings an expression may take. This led to the reframing of $(\bullet)$ and $(\circledast)$ in terms of **R/L** and **A** above.

It is less obvious how to recast occurrences of η in these terms because η

creates an effect, where (•) and (⊛) *eliminate* them. We suggest that in the presence of **R/L**, the only use for η is in exactly the scenarios spelled out above: to feed a closure operator of some sort or another. We thus propose the rules in (3.25).

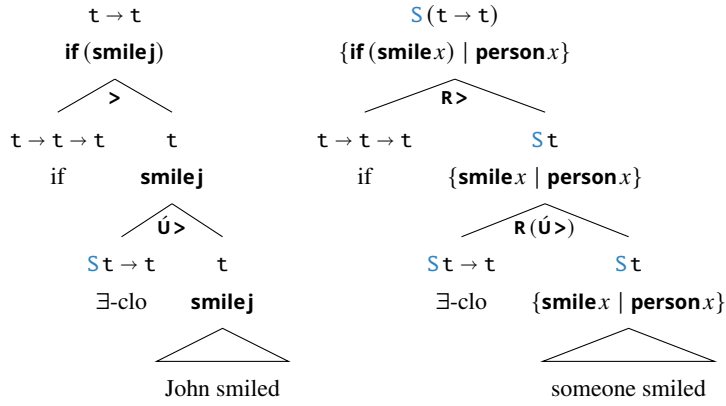
(3.25a) $\textbf{Ú} :: ((\sigma \to \sigma') \to \tau \to \omega) \to (\Theta\sigma \to \sigma') \to \tau \to \omega$
$\textbf{Ú}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, E_1\,(\eta\, a)) * E_2$

(3.25b) $\textbf{Ù} :: (\tau \to (\sigma \to \sigma') \to \omega) \to \tau \to (\Theta\sigma \to \sigma') \to \omega$
$\textbf{Ù}\,(*)\,E_1\,E_2 \coloneqq E_1 * (\lambda b.\, E_2(\eta\, b))$

These rules say that a closure operator of type $\Theta\sigma \to \sigma'$ (on the left or the right) may be combined with a prejacent of type τ whenever the function type $\sigma \to \sigma'$ could be combined with τ. Here's how: convert the closure operator into an ordinary function of type $\sigma \to \sigma'$ by composing it with η. That is, create the function that will take in an input of type σ, inject it into Θ with η, and then pass that newly created computation of type $\Theta\sigma$ into the closer operator. Then combine this function of type $\sigma \to \sigma'$ with the other daughter of type τ in whatever way(s) make sense.

Derivations of (3.22) and (3.24) using this new unit meta-combinator are shown in (3.26).

(3.26)



The complete applicative and functor rules are summarized in Figure 7.

Types:

$$
\begin{array}{rl}
\tau ::= & \mathtt{e} \mid \mathtt{t} \mid \ldots \qquad \text{Primitive types} \\
\mid & \tau \to \tau \qquad \text{Function types} \\
\mid & \ldots \qquad \ldots \\
\mid & \Sigma\tau \qquad \text{Computation types}
\end{array}
$$

Effects:

$$
\begin{array}{rl}
\Sigma ::= & \mathtt{G} \qquad \text{Reading} \\
\mid & \mathtt{W} \qquad \text{Writing} \\
\mid & \mathtt{S} \qquad \text{Indeterminacy} \\
\mid & \ldots \qquad \ldots
\end{array}
$$

Basic Combinators:

$$(\texttt{>}) :: (\alpha \to \beta) \to \alpha \to \beta \qquad \text{Forward Application}$$
$$f \texttt{ > } x \coloneqq f\,x$$

$$(\texttt{<}) :: \alpha \to (\alpha \to \beta) \to \beta \qquad \text{Backward Application}$$
$$x \texttt{ < } f \coloneqq f\,x$$

$$\vdots$$

Meta-combinators:

$$\mathtt{L} :: (\sigma \to \tau \to \omega) \to \Theta\sigma \to \tau \to \Theta\omega \qquad \text{Map Left}$$
$$\mathtt{L}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, a * E_2) \bullet E_1$$

$$\mathtt{R} :: (\sigma \to \tau \to \omega) \to \sigma \to \Theta\tau \to \Theta\omega \qquad \text{Map Right}$$
$$\mathtt{R}\,(*)\,E_1\,E_2 \coloneqq (\lambda b.\, E_1 * b) \bullet E_2$$

$$\mathtt{A} :: (\sigma \to \tau \to \omega) \to \Theta\sigma \to \Theta\tau \to \Theta\omega \qquad \text{Structured App}$$
$$\mathtt{A}\,(*)\,E_1\,E_2 \coloneqq (\lambda a \lambda b.\, a * b) \bullet E_1 \circledast E_2$$

$$\acute{\mathtt{U}} :: ((\sigma \to \sigma') \to \tau \to \omega) \to (\Theta\sigma \to \sigma') \to \tau \to \omega \qquad \text{Unit Right}$$
$$\acute{\mathtt{U}}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, E_1\,(\eta\, a)) * E_2$$

$$\grave{\mathtt{U}} :: (\sigma \to (\tau \to \tau') \to \omega) \to \sigma \to (\Theta\tau \to \tau') \to \omega \qquad \text{Unit Left}$$
$$\grave{\mathtt{U}}\,(*)\,E_1\,E_2 \coloneqq E_1 * (\lambda b.\, E_2(\eta\, b))$$

Figure 7 A type-driven grammar with functors and applicatives

3.4 Applicative typology

The grammar of Figure 7 is in some sense maximally expressive relative the applicative structure of an effect. It permits every possible agglomeration or stratification of structure, which in turn means that closure operators may capture anywhere from none to all of the effects in their prejacents.

It is perhaps worth taking stock of how the different ingredients of Figure 7 regulate this expressivity. One way to carry out this thought experiment is to investigate the capturing profile of closure operators under various ablations of the meta-combinators.

For instance, with a purely functorial grammar that includes only (•), we predict that any operator $\lightning :: \boxminus\sigma \to \tau$ will necessarily associate with exactly one effect. The reason is that while (•) suffices to generate higher-order effects, the operator $\lightning$ only knows how to process a single layer. All the others will have to be mapped over it. And if there aren't any effects, then $\lightning$ is out of luck because with only (•), there is no way to create a computation where one did not exist before.

A grammar with only (⊛) and no mapping or unit combinators is essentially the Heim and Kratzer and Hamblin grammars of Figures 5 and 6. For starters, in order for composition to be possible at all, all lexical items will need to be coerced into the computation type $\boxminus$ in the lexicon. That done, the grammar will predict that all closure operators $\lightning$ necessarily associate with at least one effect, and also necessarily capture all of their prejacent's effects. Without (•), there is no way to scope an effect over $\lightning$, and without η, there is no way to fake an effect where one is not expressed.

Excluding only η yields a grammar where all closure operators associate with at least one effect, but do so selectively ((•) allows all but one effect to optionally pass over the closure). On the flipside, excluding only (⊛) means all closure operators associate with at most one effect (η can create computations at will, simulating effects where none are expressed, but no (⊛) means that each effect creates a distinct computational layer, of which $\lightning$ can target just one). Finally, excluding only (•) does not reduce the expressivity of the full grammar in Figure 7, provided that free insertions of η are permitted. This is thanks to the equivalence in (3.6), which ensures the mapping operations can always be simulated by a composition of η and (⊛).

All this to say, just because a type constructor is mathematically applicative doesn't mean that the particular instance(s) of its associated (⊛) and η combinators need be included in a grammar or fragment. It is highly likely that different natural language operators evince different empirical capturing properties, so it is useful to know what happens when these algebraic knobs are twiddled.

3.5 Commutative and non-commutative effects

In this chapter we've concentrated on G and S effects as canonical examples of applicativity in linguistics. These two effects have in common that they are **commutative**: when using **A** to combine $E_1 :: \Theta\,(\sigma \to \tau)$ and $E_2 :: \Theta\,\sigma$, it doesn't matter which daughter is on the left and which is on the right. That is, for any E_1 and E_2 with these types, we have the following equivalence.

(3.27) $\quad \mathbf{A}\,(>)\,E_1\,E_2 = \mathbf{A}\,(<)\,E_2\,E_1$

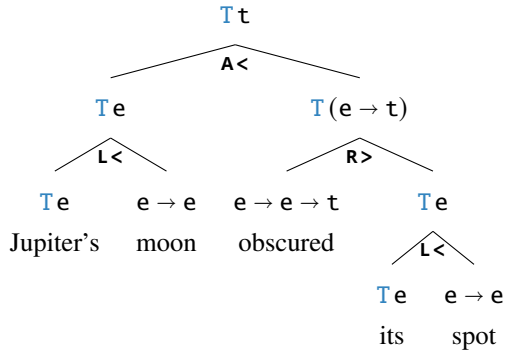
This is because the $(\circledast)$ instance for G simply passes an incoming environment in to both computations E_1 and E_2 before applying the one to the other. And the $(\circledast)$ instance for S takes the full cross-product of its daughters, selecting every element from the one and every element from the other independently.

But not all effects are like this. As a segue into the next section, we draw attention to two such non-commutative effects. The first, T, is defined in (3.28).

(3.28)
$$\begin{aligned} \mathtt{T}\,\alpha &::= \mathtt{s} \to (\alpha \times \mathtt{s}) \\ \eta\,x &:= \lambda i.\,\langle x,\, i\rangle \\ F \circledast X &:= \lambda i.\,\langle f\,x,\, k\rangle, \textbf{ where } \langle f,\, j\rangle = F\,i \\ &\qquad\qquad\qquad\qquad\quad \langle x,\, k\rangle = X\,j \end{aligned}$$

The constructor T models a transition from one context s to another, computing an α along the way. This is a simplified version of the sort of state-updating denotations often seen in dynamic semantics. For instance, if we identify the context type s with a list of mentioned entities (Eijck, 2001; Vermeulen, 1993), then we might imagine that the denotations of names add referents to the outgoing context, while the denotations of pronouns read antecedents from the incoming context. Sequencing an expression containing a name and a pronoun, *in that order*, should result in a kind of dynamic binding.

(3.29)



Importantly, reversing the order of the daughters would reverse which computation passes its output in to the other. Compare $\mathbf{A}\,(>)\,E_1\,E_2$ and $\mathbf{A}\,(<)\,E_2\,E_1$ below:

$$\text{(3.30a)}\quad \mathbf{A}\,(>)\,E_1\,E_2 \coloneqq \lambda i.\,\langle f\,x,\,k\rangle,\ \textbf{where}\ \langle f,\,j\rangle = E_1\,i \quad \langle x,\,k\rangle = E_2\,j$$

$$\text{(3.30b)}\quad \mathbf{A}\,(<)\,E_2\,E_1 \coloneqq \lambda i.\,\langle f\,x,\,k\rangle,\ \textbf{where}\ \langle x,\,j\rangle = E_2\,i \quad \langle f,\,k\rangle = E_1\,j$$

In the first case, E_1 is evaluated at the input state i, and its output j is passed in as input to E_2. In the second case, E_2 is evaluated at the input i, and its output j is passed in as input to E_1. The only way for E_1 to influence the state that E_2 is evaluated against is for it to come first.

The second non-commutative effect, defined by C in (3.31), models Generalized Quantifiers over the domain of α.

$$\begin{aligned}\text{(3.31)}\qquad \mathsf{C}\,\alpha &::= (\alpha \to \mathtt{t}) \to \mathtt{t}\\ \eta\,x &\coloneqq \lambda c.\,c\,x\\ F \circledast X &\coloneqq \lambda c.\,F\,(\lambda f.\,X\,(\lambda x.\,c\,(f\,x)))\end{aligned}$$

Here, "applying" one quantifier $F :: \mathsf{C}\,(\sigma \to \tau)$ to another $X :: \mathsf{C}\,\sigma$ means passing the latter in as the part of the scope of the former. Reversing the order of evaluation amounts to inverting their scopes, as seen in (3.32) and therefore (often) the resulting denotation.

$$\text{(3.32a)}\quad \mathbf{A}\,(>)\,E_1\,E_2 \coloneqq \lambda c.\,E_1\,(\lambda f.\,E_2\,(\lambda x.\,c\,(f\,x)))$$

$$\text{(3.32b)}\quad \mathbf{A}\,(<)\,E_2\,E_1 \coloneqq \lambda c.\,E_2\,(\lambda x.\,E_1\,(\lambda f.\,c\,(f\,x)))$$

3.6 Implementing applicative effects in the type-driven interpreter

Adding applicativity to our interpreter will follow exactly the same format as adding functoriality did in Chapter 2. We start again by adding modes the structure-preserving application **A** and unit rules **Ú**/**Ù**. Again because these are meta-combinators, they are parameterized to the modes that would combine the relevant underlying types.

```
data Mode
  = FA | BA | PM        -- etc
  | MR Mode | ML Mode   -- map right and map left
  | AP Mode             -- structured application
  | UR Mode | UL Mode   -- unit right and unit left
```

For the type logic, we need a new predicate to characterize which effects have applicative instances. The only case to watch out for is that of `W`, which is parameterized by the type of data that it stores in its second dimension. In order to combine two `W` computations that both have stored, secondary data, we have to make sure that the supplemental values can be, in some sense, merged. This is checked with the predicate `monoid`. For this simple, illustrative system, the only type that is monoidal is `T`, where merger is understood to be conjunction. All of the other functorial effects presented in this Element are also applicative, so the elsewhere clause of this function is extensionally equivalent to `functor`.

```
functor, applicative :: EffX -> Bool
functor     _      = True
applicative (WX s) = monoid s
applicative f      = functor f && True

monoid :: Ty -> Bool
monoid T = True
monoid _ = False
```

Extending the combination function `combine` follows the same recursive logic as in Chapter 2. When combining a left and right daughter, we start by including any of the earlier basic and functorial operations, and then we look to add applicative ones if possible. For instance, if both daughters are computation types `Comp f s` and `Comp g t` with the same applicative effect (`f == g, applicative f`), then try combining the underlying types `s` and `t`. For every way `(op, u)` that those underlying types can be combined, build a new composite mode of combination `AP op` with combined type `Comp f u`. The unit rules are similar, trying to combine underlying types, and building on the results.

```
combine :: Ty -> Ty -> [(Mode, Ty)]
combine l r =
  -- see if any basic modes of combination work
  modes l r
```

```
  -- if the right daughter is functorial, try to map over it
  ++ addMR l r
  -- if the left daughter is functorial, try to map over it
  ++ addML l r
  -- if both daughters are applicative, try structured application
  ++ addAP l r
  -- if the left daughter closes an applicative effect,
  -- try to purify the right daughter
  ++ addUR l r
  -- if the right daughter closes an applicative effect,
  -- try to purify the left daughter
  ++ addUL l r

addAP l r = case (l, r) of
  (Comp f s, Comp g t) | f == g, applicative f
    -> [ (AP op, Comp f u) | (op, u) <- combine s t ]
  _ -> [                                            ]

addUR l r = case l of
  Comp f s :-> s' | applicative f
    -> [ (UR op, u) | (op, u) <- combine (s :-> s') r ]
  _ -> [                                              ]

addUL l r = case r of
  Comp f t :-> t' | applicative f
    -> [ (UL op, u) | (op, u) <- combine l (t :-> t') ]
  _ -> [                                              ]
```

4 Monads

4.1 Effects inside effects

Up to this point we have ignored the internal compositional details of various noun phrases, writing things like:

Expression	Type	Denotation
the cat	$\mathtt{M}\,\mathtt{e}$	x if **cat** $= \{x\}$ else #
a cat	$\mathtt{S}\,\mathtt{e}$	$\{x \mid \mathbf{cat}\,x\}$
every cat	$\mathtt{C}\,\mathtt{e}$	$\lambda c.\, \neg\exists x.\, \mathbf{cat}\,x \wedge c\,x$
...	...	...

Consider then the semantics of a determiner. Each one creates a specific sort of computation, the particulars of which depend on the property that restricts it. Take the indefinite article 'a' for example. It creates an indeterminate computation by sifting each of its restrictor's witnesses into an isolated compositional thread. Or the definite article 'the', which introduces partiality by creating a computation that might crash, depending on the property it is handed. And so on for the others.

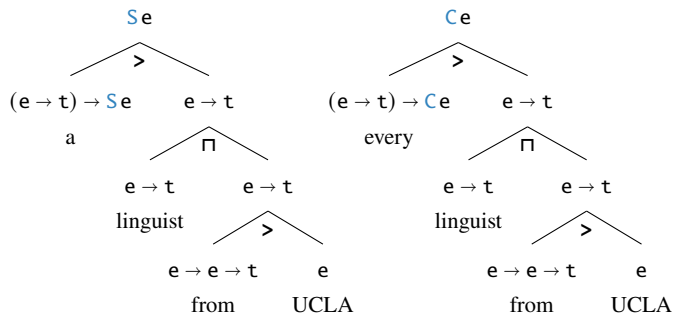
The natural denotations for these sorts of operations are functions from properties to computations.

Expression	Type	Denotation
the	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{M}\,\mathtt{e}$	$\lambda P.\, x$ if $P = \{x\}$ else #
a	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{S}\,\mathtt{e}$	$\{x \mid \mathbf{cat}\,x\}$
every	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{C}\,\mathtt{e}$	$\lambda P \lambda c.\, \neg\exists x.\, P\,x \wedge c\,x$
...	$(\mathtt{e} \to \mathtt{t}) \to \Theta\,\mathtt{e}$	...

Types like these are the converses of the closure operators in the previous section. Where a closure operator $\lightning :: \Theta\,\mathtt{t} \to \mathtt{t}$ *reduces* an effect $\Theta\,\mathtt{t}$ to a value $\mathtt{t}$, a determiner *generates* an effect $\Theta\,\mathtt{e}$ from a value $(\mathtt{e} \to \mathtt{t})$. In Category Theoretic settings, types with this general shape — $\sigma \to \Theta\,\tau$ — are known as **Kleisli arrows**, and types with the shape of closure operators — $\Theta\,\sigma \to \tau$ — known as **co-Kleisli**

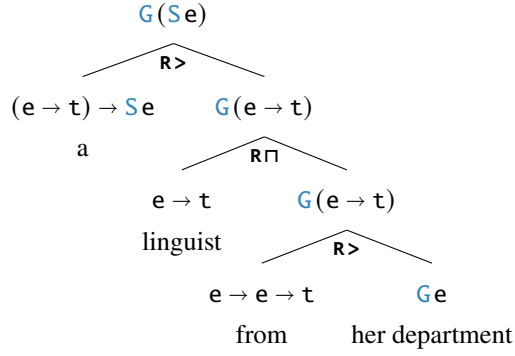
arrows. Simple examples of composition with these types are given in (4.1)

(4.1)



What happens when the restrictor itself denotes a computation, as in (4.2)? Nothing particularly special. The modes of combination at the nodes of the noun phrase must be mapped over the new effect, but are otherwise unchanged.

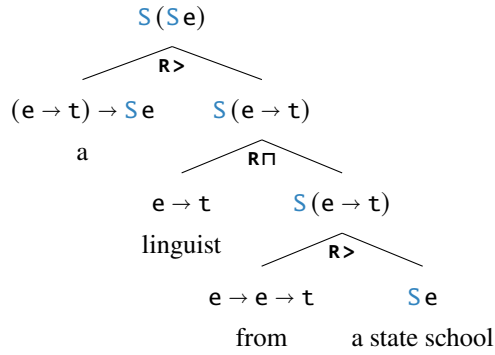
(4.2)



Of note, however, is that the phrase is *not* ambiguous. Only one layering of the two effects is possible. The restrictor's effect must take wider scope. There is simply no other way to combine the pieces.

This is true even when the computation in the restrictor is of the same ilk as that generated by the determiner.

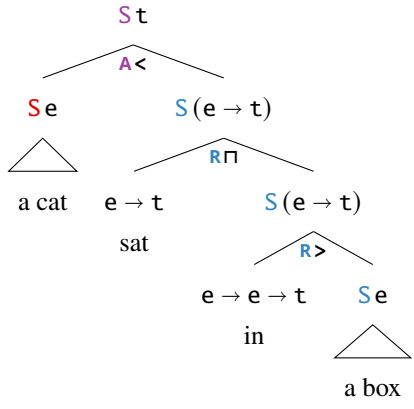
(4.3)



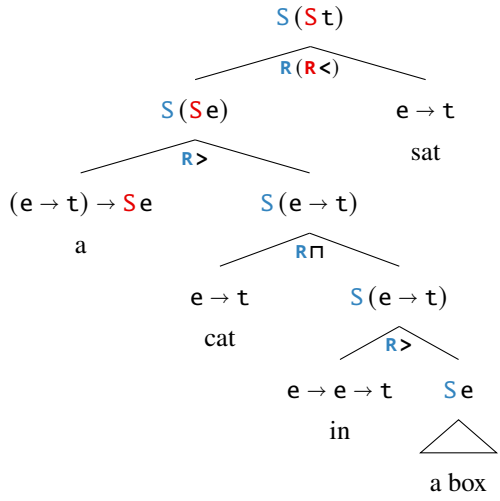
The determiner needs to get at the value(s) computed by the restrictor, the witnesses of that underlying ordinary property type (e → t). With the algebraic ingredients developed thus far, the way to do that — the *only* way to do that — is to map over the effects of the restrictor's computation, as in (4.3). This means that the resulting denotation is necessarily higher-order, *even though the* S *effect is applicative*.

Compare the derivations in (4.4a) and (4.4b). The un-nested configuration in (4.4a) has a now familiar applicative combination, merging the cat and box alternatives into one set of propositions. But the nested configuration in (4.4b) has no such combination. We are necessarily left with a set of sets of propositions. For each box, we compute the set containing one proposition per cat.

(4.4a)

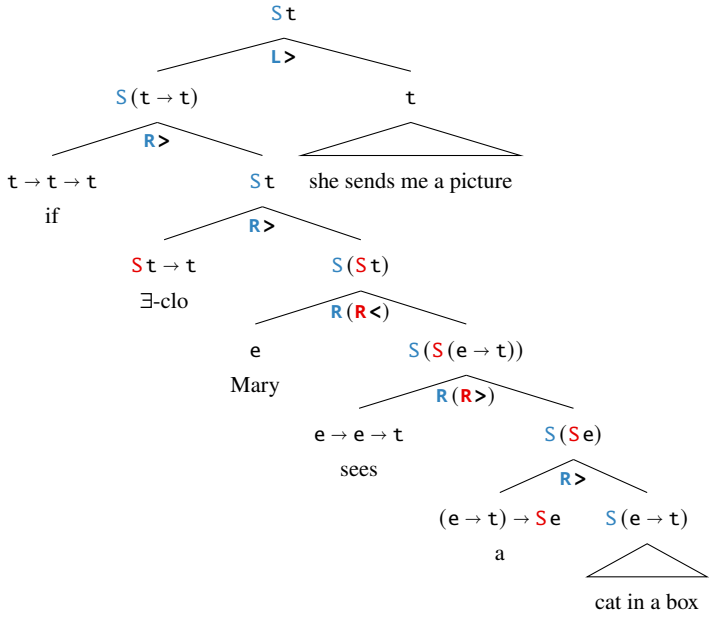


(4.4b)



This predicts that an otherwise **unselective** closure operator will necessarily fail to capture effects that happen to be nested in the arguments of Kleisli arrows. For instance, assuming as in Chapter 3 that the antecedent of a conditional is associated with an existential closure operator, the current state of affairs predicts that a nested indefinite will *necessarily* take exceptional scope over the conditional.

(4.5)



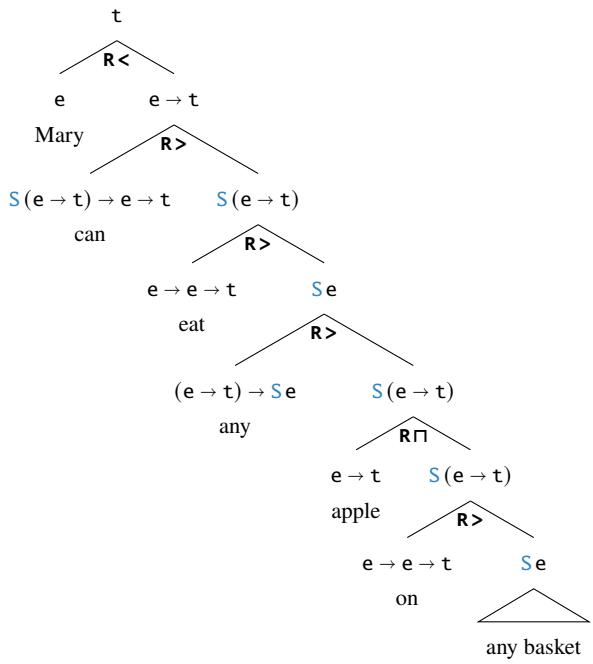
Certainly for English indefinites at least, this prediction is false. The particular problem might be solved by adding more ∃-clo operators in the antecedent, but that strategy is less sensible when the closure operator is a lexical item that associates unselectively with effects. For illustration, consider the entry in (4.6) defining 'can' as an alternative-sensitive modal operator (see, e.g., Goldstein 2019 for discussion of analyses along these lines). Given a set of options m determined by S-generators in its prejacent, the modal ensures that every proposition in m is a live possibility.

(4.6) $\quad \text{can} :: \mathsf{S}(\mathtt{e} \to \mathtt{t}) \to \mathtt{e} \to \mathtt{t}$

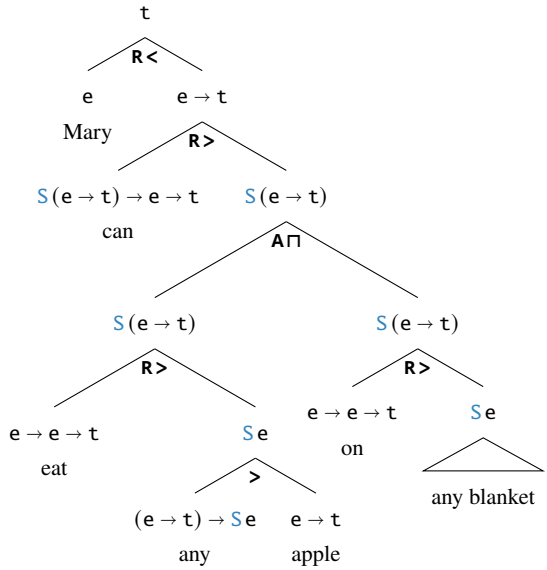
$\quad [\![\text{can}]\!] \coloneqq \lambda m \lambda x.\ \bigwedge\{\Diamond(Px) \mid P \in m\}$

As things stand, a difference is predicted in the possible readings of (4.7a) and (4.7b). Only the latter is predicted to confer total freedom to Mary in her choice of apples and blankets. The former only grants universal permission to the apples of a particular blanket. This is empirically disappointing, since in reality it doesn't matter whether it's the apples or Mary that's on the blanket. Both parses have unselective readings.

(4.7a)



(4.7b)



The point is not to argue for any particular analysis of English free choice semantics or indefinite scope delimitation. The point is that if there is *any* operator that can associate with nested effects just the same as it can with un-nested effects, that pattern is beyond the expressive reach of the current grammar.

And such nestings do not only occur in the arguments of determiners. They are liable to pop up any time a Kleisli arrow appears. For instance, consider the abstraction operator commonly used to regulate movement and binding, defined with an effect-theoretic type in (4.8)

(4.8) $\lambda_n :: \mathsf{G}\beta \to \mathsf{e} \to \mathsf{G}\beta$

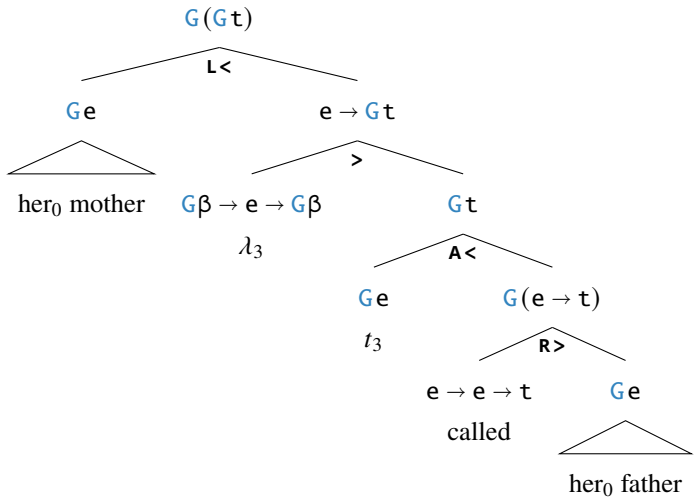
$[\![\lambda_n]\!] := \lambda m \lambda x \lambda g.\, m\, g^{n \mapsto x}$

Given an environment-dependent meaning $m :: \mathsf{G}\beta$, the abstraction λ_n binds an argument x to the nth coordinate of the environment that m is evaluated in. Any pro-forms that access this coordinate will therefore resolve to x when evaluated.

The return type of λ_n is a Kleisli arrow: $\mathsf{e} \to \mathsf{G}\beta$. What happens when the specifier of the λ_n is itself environment-dependent, as in (4.9)? For no good reason, we are forced to end up with a denotation that needs *two* environments in order to return a truth value. The first, outer environment will be used to value the subject, 'her$_0$ mother', and the second, inner environment used to value the

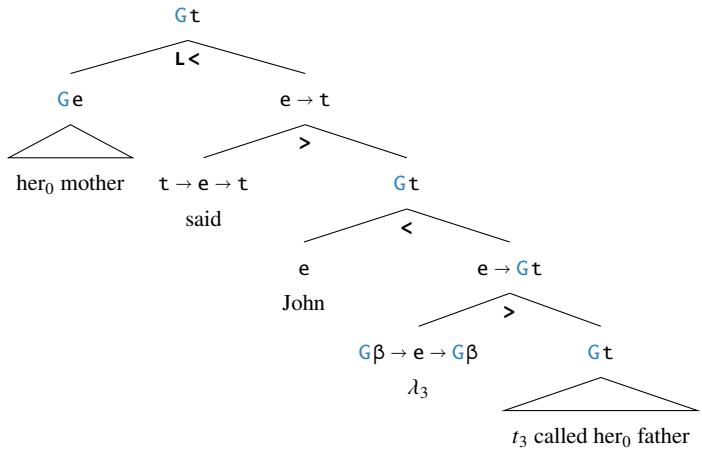
object, 'her$_0$ father', even though the pronoun is the same in both cases.

(4.9)

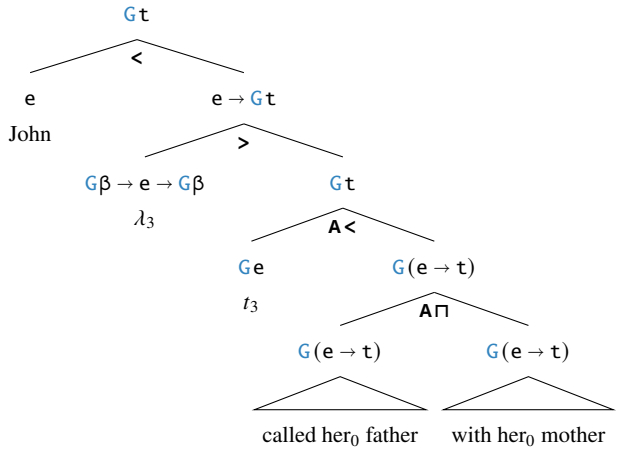


Compare this with what happens when the two pronouns occur in nearly any other configuration. With 'her$_0$ mother' just a little bit higher or lower, the environment-sensitivities of the two anaphoric expressions can be merged as usual.

(4.10)



(4.11)



What is needed then is a way to combine a Kleisli arrow $k :: \sigma \to \Theta\,\tau$ with a computation $m :: \Theta\,\sigma$ so as to produce a composite computation $\Theta\,\tau$ that amalgamates the effects of m and k. It is not hard to see that such combinations are easy to define when Θ is G or S, which immediately irons out the wrinkles in the examples above. And it turns out that analogous combinations are in fact definable for all of the computation types in Table 2. Moreover, the various combinators share important algebraic relationships with $(\bullet)$, η, and $(\circledast)$, which we describe in the next section.

4.2 Flattening effects

Let's start with G. The task is to find a way of putting something of type $k :: \sigma \to \mathsf{G}\,\tau$ together with something of type $m :: \mathsf{G}\,\sigma$. The obvious candidate — indeed the only polymorphic function with this type — is given in (4.12).

(4.12) $(\star_{\mathsf{G}}) :: (\sigma \to \mathsf{G}\,\tau) \to \mathsf{G}\,\sigma \to \mathsf{G}\,\tau$

$k \star_{\mathsf{G}} m \coloneqq \lambda g.\, k\,(m\,g)\,g$

Notice that this just permutes the arguments of the corresponding $(\circledast)$ operation on G, which is fitting since $(\star)$ and $(\circledast)$ differ only in the type of their first argument: $\sigma \to \mathsf{G}\,\tau$ vs. $\mathsf{G}\,(\sigma \to \tau)$. And the only difference between these types is whether the environment argument corresponding to G comes before or after the ordinary argument corresponding to σ.

Let's try S. We're now seeking an operation $(\star_{\mathsf{S}})$ to combine a function $k :: \sigma \to \mathsf{S}\,\tau$ with an argument $m :: \mathsf{S}\,\sigma$. After a bit of thought, the following

emerges as a natural choice.

(4.13) $(\star_{\mathsf{S}}) :: (\sigma \to \mathsf{S}\,\tau) \to \mathsf{S}\,\sigma \to \mathsf{S}\,\tau$
$k \star_{\mathsf{S}} m \coloneqq \bigcup\{k\,x \mid x \in m\}$

This time there are other functions we could imagine doing the job. The big union, for instance, could just as well be swapped out for a big intersection, as far as the types are concerned. But there is an important sense in which the definition in (4.13) preserves all of the effect structure of m and k. A grand intersection would likely throw out alternatives generated by at least one of m and k, certainly not something we'd want from a mode of combination.

One way to formalize the sense in which $(\star)$ does not add or lose any information about the effects generated by m and k is to note that for both G and S, we have the following equivalences.

(4.14a)
$$\begin{aligned}
&\lambda x.\, k \star_{\mathsf{G}} (\eta_{\mathsf{G}}\, x)\\
&= \lambda x.\, k \star_{\mathsf{G}} (\lambda h.\, x)\\
&= \lambda x \lambda g.\, k\,((\lambda h.\, x)\,g)\,g\\
&= \lambda x \lambda g.\, k\,x\,g\\
&= k
\end{aligned}$$

(4.14b)
$$\begin{aligned}
&\eta_{\mathsf{G}} \star_{\mathsf{G}} m\\
&= (\lambda x \lambda h.\, x) \star_{\mathsf{G}} m\\
&= \lambda g.\, (\lambda x \lambda h.\, x)\,(m\,g)\,g\\
&= \lambda g.\, m\,g\\
&= m
\end{aligned}$$

(4.15a)
$$\begin{aligned}
&\lambda x.\, k \star_{\mathsf{S}} (\eta_{\mathsf{S}}\, x)\\
&= \lambda x.\, k \star_{\mathsf{S}} \{x\}\\
&= \lambda x.\, \{z \mid a \in \{x\},\ z \in k\,a\}\\
&= \lambda x.\, \{z \mid z \in k\,x\}\\
&= k
\end{aligned}$$

(4.15b)
$$\begin{aligned}
&\eta_{\mathsf{S}} \star_{\mathsf{S}} m\\
&= (\lambda x.\, \{x\}) \star_{\mathsf{S}} m\\
&= \{z \mid a \in m,\ z \in (\lambda x.\, \{x\})\,a\}\\
&= \{a \mid a \in m\}\\
&= m
\end{aligned}$$

The reductions in (4.14a) and (4.15a) guarantee that no information in k is lost when it is combined via $(\star)$; since η creates a trivial computation, the only effects in $k \star \eta\,x$ should come from k. The reductions in (4.14b) and (4.15b) guarantee that no information is added to m when it is combined via $(\star)$; again the reason is that since η doesn't do anything interesting, the only modification to m would have to come from $(\star)$.

An applicative functor for which there is such a well-behaved $(\star)$ operator is known as a **monad**. Formally, to be well-behaved the operator should respect the laws in (4.16). The first two equations are just generalizations of the facts observed above. We discuss the significance of **Associativity** in Section 4.3.

(4.16) **Left Identity:** $\eta \star m = m$

Right Identity: $k \star \eta x = k\,x$

Associativity: $k \star (c \star m) = (\lambda x.\, k \star c\,x) \star m$

In Haskell, the $(\star)$ operation is spelled `(=<<)`. In practice, it is often convenient to work with a version of $(\star)$ that takes its arguments in the opposite order:

(4.17) $m \gg\!= k \coloneqq k \star m$

Indeed, in Haskell, it is this flipped version that the standard `Monad` type class implements, where it is given the name `(>>=)`, pronounced "bind". Obviously `(=<<)` and `(>>=)` are interdefinable. Also, for historical reasons, the `pure` operation guaranteed by the applicativity of the constructor is redundantly specified in the `Monad` class, where it is called `return`.

```
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b

  (=<<) :: (a -> f b) -> f a -> f b
  k =<< m = m >>= k

  return :: a -> f a
  return = pure
```

One thing to notice about the $(\star)$ operations defined in (4.12) and (4.13) is that they both implicitly incorporate the corresponding definitions of $(\bullet)$ for their types. This is certainly easiest to see in (4.13), which is clearly a mapping of k over m — $k \bullet_S m \coloneqq \{k\,x \mid x \in m\}$ — followed by a flattening with $\bigcup$. We might just as well have written $k \star_S m \coloneqq \bigcup(k \bullet_S m)$.

Upon inspection, (4.12) can also be seen as a mapping of k over m — $k \bullet_G m \coloneqq \lambda g.\, k\,(m\,g)$ — followed by a sort of flattening, the re-use of the argument g. The traditional name for this argument-duplicating operation is $\mathbf{W} \coloneqq \lambda M \lambda g.\, M\,g\,g$. Using this, we might just as well have written (4.12) as $k \star_G m \coloneqq \mathbf{W}(k \bullet_G m)$.

And in general, every monadic $(\star) :: (\alpha \to \Theta\beta) \to \Theta\alpha \to \Theta\beta$ is a composition of an effect-mapping operation $(\bullet) :: (\alpha \to \beta) \to \Theta\alpha \to \Theta\beta$ and an effect-flattening operation $\mu :: \Theta(\Theta\beta) \to \Theta\beta$. That is, for every monad Θ, there is a μ such that:

(4.18) $k \star m = \mu(k \bullet m)$

It therefore suffices to define the relevant $\mu :: \theta\,(\theta\,\beta) \to \theta\,\beta$ that would make the derived $(\star)$ law-preserving.

Haskell's customary name for μ is `join`. For quite obscure technical and historical reasons, `join` is left out of the standard `Monad` type class, but in principle it could have been defined as follows.

```
class Applicative f => Monad f where
  join :: f (f a) -> f a

  return :: a -> f a
  return = pure

  m >>= k = join (fmap k m)
  k =<< m = m >>= k
```

Finally, we should point that only functors `f` which are applicative can be monads, as implied by the class hierarchy `Applicative f => Monad f`. In other words, every monad is an applicative functor, and therefore a functor. In fact, the applicative and functorial combinators for a type can be derived from $(\gg\!=)$ and η by the recipes in (4.19a) and (4.19b).

(4.19a) $\quad k \bullet m = m \gg\!= \lambda x.\, \eta\,(k\,x)$

(4.19b) $\quad F \circledast X = F \gg\!= \lambda f.\, X \gg\!= \lambda x.\, \eta\,(f\,x)$

Or in Haskell:

```
fmap k m  = m  >>= \x -> return (k x)
mf <*> mx = mf >>= \f -> mx >>= \x -> return (f x)
```

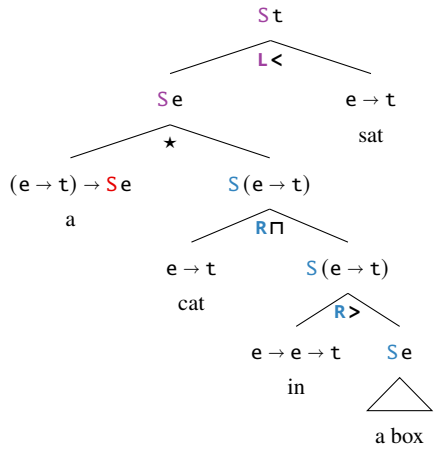
As in Chapter 3, it turns out that all of the constructors in Table 2 are monadic. We provide standard definitions of the monad instances for these effects in Appendix A4. Incorporating $(\star)$ into grammar fragments presents the same options as in the preceding sections, described below.

4.2.1 Flattening in the grammar

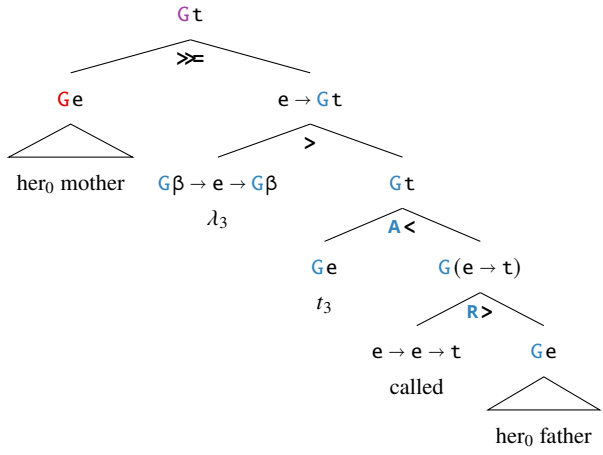
Of course, since the nodes that caused compositional trouble in (4.4b)/(4.9) have exactly the types that $(\star)$ and $(\gg\!=)$ are suited to combine, those immediate problems would be resolved just by adding these two operators to the inventory of combinatory modes.

This would provide for derivations as in (4.20) and (4.21).

(4.20)



(4.21)

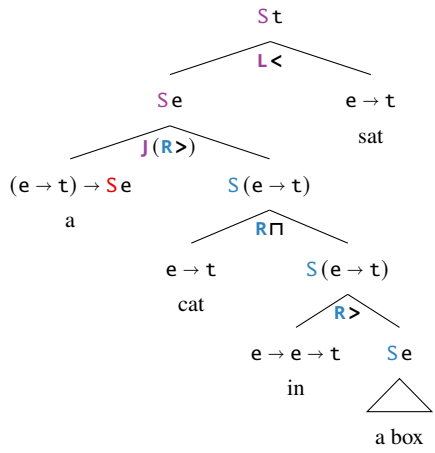


But since the grammar we have developed so far already has robust resources for mapping left or right with forward or backward application, we might as well make use of the equivalence in (4.18). This way we add only one meta-rule for exploiting the monadic nature of effects, defined in (4.22). When the parameter $(*)$ to this rule is **R>**, the result is equivalent to $(\star)$, and when the parameter is **L<**, the result is equivalent to $(\gg\!=)$.

(4.22) $\quad \mathbf{J}\,(*)\,E_1\,E_2 := \mu\,(E_1 * E_2)$

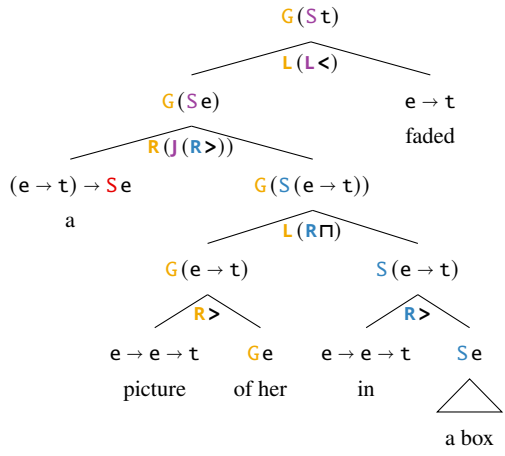
Thus we would derive (4.20), for instance, as in (4.23) instead.

(4.23)



And being a mode of combination, the **J** rule may itself appear as part of a parameter to other meta-combinators like **R**, **L**, and **A**. This guarantees that incidental occurrences of other kinds of effects can continue to bubble up even as monadic effects are ironed out below them. The derivation in (4.24) provides an example.

(4.24)



The entire grammar, extended with **J**, is presented in Figure 8.

Types:

$$\begin{array}{lll} \tau ::= \mathtt{e} \mid \mathtt{t} \mid \ldots & & \text{Primitive types} \\ \quad \mid \tau \to \tau & & \text{Function types} \\ \quad \mid \ldots & & \ldots \\ \quad \mid \Sigma\tau & & \text{Computation types} \end{array}$$

Effects:

$$\begin{array}{lll} \Sigma ::= \mathtt{G} & & \text{Reading} \\ \quad \mid \mathtt{S} & & \text{Indeterminacy} \\ \quad \mid \ldots & & \ldots \end{array}$$

Basic Combinators:

$$\begin{array}{ll} (\mathtt{>}) :: (\alpha \to \beta) \to \alpha \to \beta & \text{Forward Application} \\ f \mathtt{>} x \coloneqq f\,x & \\ (\mathtt{<}) :: \alpha \to (\alpha \to \beta) \to \beta & \text{Backward Application} \\ x \mathtt{<} f \coloneqq f\,x & \\ \ldots & \end{array}$$

Meta-combinators:

$$\begin{array}{ll} \mathbf{L} :: (\sigma \to \tau \to \omega) \to \Theta\sigma \to \tau \to \Theta\omega & \text{Map Left} \\ \mathbf{L}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, a * E_2) \bullet E_1 & \\ \mathbf{R} :: (\sigma \to \tau \to \omega) \to \sigma \to \Theta\tau \to \Theta\omega & \text{Map Right} \\ \mathbf{R}\,(*)\,E_1\,E_2 \coloneqq (\lambda b.\, E_1 * b) \bullet E_2 & \\ \mathbf{A} :: (\sigma \to \tau \to \omega) \to \Theta\sigma \to \Theta\tau \to \Theta\omega & \text{Structured App} \\ \mathbf{A}\,(*)\,E_1\,E_2 \coloneqq (\lambda a \lambda b.\, a * b) \bullet E_1 \circledast E_2 & \\ \acute{\mathbf{U}} :: ((\sigma \to \sigma') \to \tau \to \omega) \to (\Theta\sigma \to \sigma') \to \tau \to \omega & \text{Unit Right} \\ \acute{\mathbf{U}}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, E_1\,(\eta\, a)) * E_2 & \\ \grave{\mathbf{U}} :: (\sigma \to (\tau \to \tau') \to \omega) \to \sigma \to (\Theta\tau \to \tau') \to \omega & \text{Unit Left} \\ \grave{\mathbf{U}}\,(*)\,E_1\,E_2 \coloneqq E_1 * (\lambda b.\, E_2\,(\eta\, b)) & \\ \mathbf{J} :: (\sigma \to \tau \to \Theta\,(\Theta\omega)) \to \sigma \to \tau \to \Theta\omega & \text{Join} \\ \mathbf{J}\,(*)\,E_1\,E_2 \coloneqq \mu\,(E_1 * E_2) & \end{array}$$

Figure 8 A type-driven grammar with monads

4.3 Scope and ★-ing

4.3.1 LFs and abstraction

Monads have played an important role in attempts to provide denotational semantics for **imperative programming languages**. These are languages that include commands for the sorts of actions described in Chapter 1: (re-)assigning values to variables, throwing errors, starting loops, etc. Such commands are described as denotating computations in exactly the way that the expressions in Table 2 have been here. And sequences of two such commands, one of which depends on the value computed by the other, are given meanings in terms of (★).

Functional programming languages, unlike imperative languages, and unlike natural languages, make these denotational mechanisms explicit in the syntax of expressions. In effect, everything that is packed into the modes of combination in Figure 8, and often left implicit in an imperative language, must be typed out as part of the program itself in a functional language like Haskell. This means there are a lot of explicit `>>=`s and `fmap`s gluing everything together. In fact, `>>=` has played such an outsized role in structuring Haskell programs that it has its own syntax called `do`-notation.

`do`-blocks represent sequences of monadic actions, chained together by `>>=`. They are intended to resemble an imperatively structured program design while maintaining referential transparency. For instance:

```
s = do x <- m
       y <- o x
       return (p y)
```

1. Compute `m` to get a value `x`
2. Pass `x` to `o` to compute a value `y`
3. Pass `y` to `p` and box up the result

This block is mechanically "de-sugared" by the compiler into a right-nested sequence of `bind`s:

```
s = m >>= (\x -> o x >>= (\y -> return (p y)))
```

Essentially, each `v <- m` `...` is translated as `m >>= (\v -> ...)`. This means that in a `do`-block:

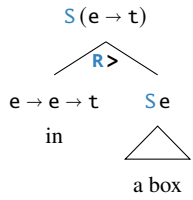
- Every expression to the right of `<-` has to have type $\theta\,\alpha$, for some monad θ
- And they all have to be *the same* monad θ
- The last line has to be an expression of type $\theta\,\varsigma$ for some type ς
- The whole block will then have type $\theta\,\varsigma$

Like everything in Haskell, `do`-blocks are just expressions. They can be used anywhere that any other expression of the same type might be used. In particular, since every `do`-block defines a computation of type $\theta\subset$ for some monad θ, they may themselves occur on the right side of a `<-` in a larger `do`-block. For instance, taking advantage of the equivalences in (4.19a), we might write the **Composition Law** for functors — $f \bullet (g \bullet M) = (f \circ g) \bullet M$ — as an equivalence between programs, as in (4.25).

(4.25)
```
do y <- do x <- m
           return (g x)
   return (f y)
```
=
```
do x <- m
   return (f (g x))
```

Now take a look at the derivation in (4.4), starting with the most embedded constituent.

(4.26)

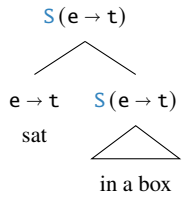


(4.27)
```
do x <- a box
   return (in' x)
```

Following the definition of **R**, this constituent is computed by the program `fmap in' (a box)`. Given the equivalence in (4.19a), this could just as well be expressed as `a box >>= \x -> return (in' x)`. Written in `do`-notation, this is the program in (4.27).

Repeating this translation at the next constituent up delivers the program in (4.29), where `(&)` is (⊓). And given the functor law spelled out in (4.25), this is equivalent to (4.30).

(4.28)

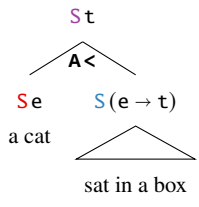


(4.29)
```
do p <- do x <- a box
           return (in' x)
   return (sat & p)
```

(4.30)
```
do x <- a box
   return (sat & in' x)
```

Finally, using the monadic encoding of (⊛) in (4.19b), the top level constituent is computed by the program in (4.32), which, again given the functor law in (4.25), is equivalent to (4.33).

(4.31)



(4.32)

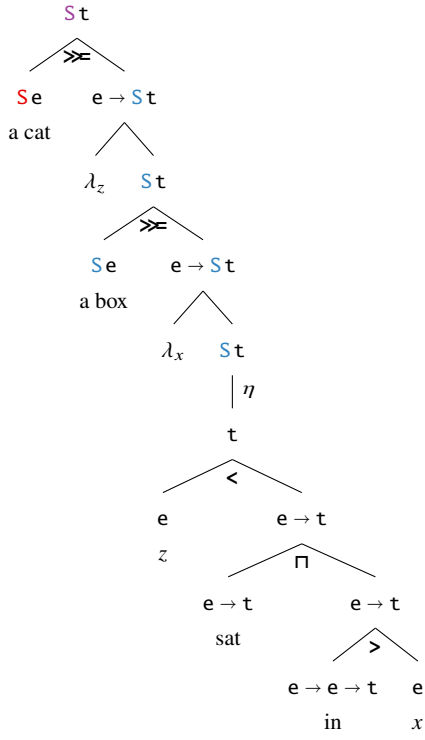
```
do z <- a cat
   p <- do x <- a box
           return (sat & in' x)
   return (p z)
```

(4.33)

```
do z <- a cat
   x <- a box
   return ((sat & in' x) z)
```

De-sugaring the `do`-notation into `>>=`s, and translating this program back into a tree yields the derivation in (4.34).

(4.34)



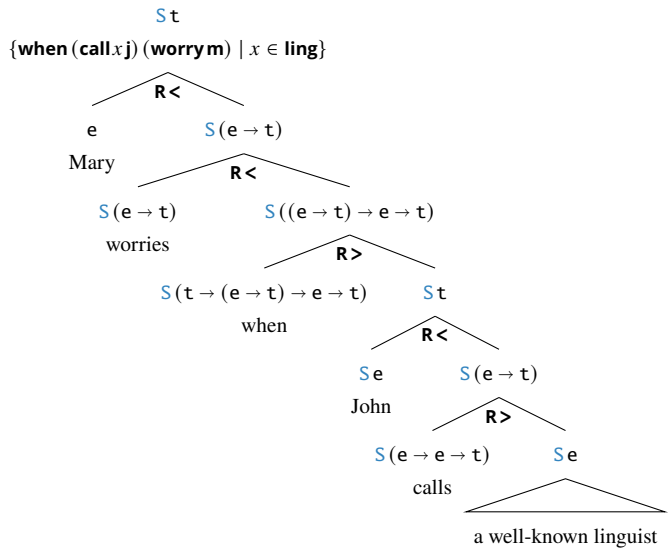
Linguists may be struck by how much this looks like Quantifier Raising. All of the properly computational expressions are raised from their argument

positions. Those positions are instead filled with variables that are abstracted over where the extra-ordinary constituent lands, forming a kind of "scope" for the computation. But these computations are not quantificational; they do not take scopes as arguments. Instead, the enriched content and its contination are combined via (⋆).

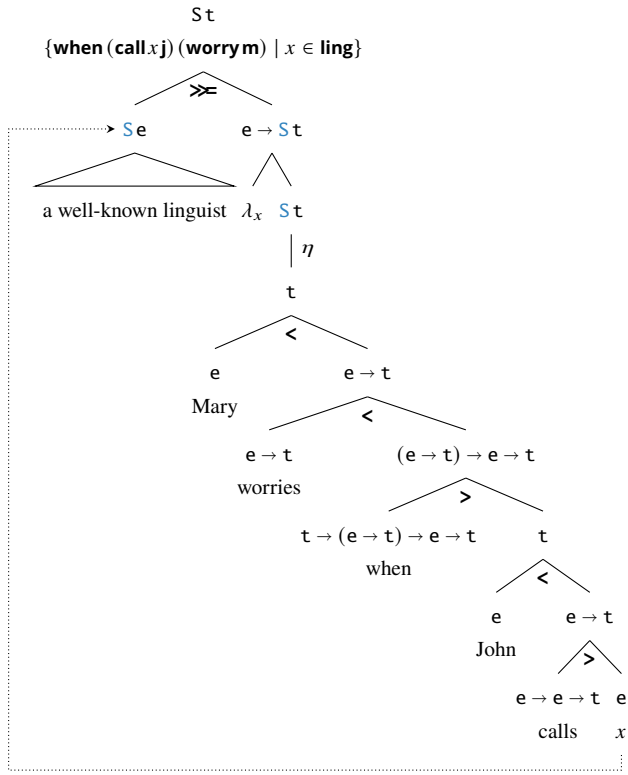
In this sense, Haskell's `do`-notation is very much like the linguist's LF. Content that cannot be interpreted *in situ* via the basic modes of combination is moved out of the way, leaving a named trace as a placeholder. All of the ordinary "business logic" of the derivation is performed with this variable. At the top, the results of this ordinary calculation are folded over the computational structure of the rich expression.

Importantly, the monad laws guarantee that this transformational derivation and the original, *in situ* derivation in (4.4) are equivalent. And indeed such equivalences will hold for any monadic effect, not just S. Also importantly, the *ex situ* and *in situ* equivalence extends to cases of **exceptional scope**. Repeating the process in (4.26)–(4.34), it's easy to see that the derivation in (4.35) is algebraically equivalent to the island-violating derivation in (4.36).

(4.35)



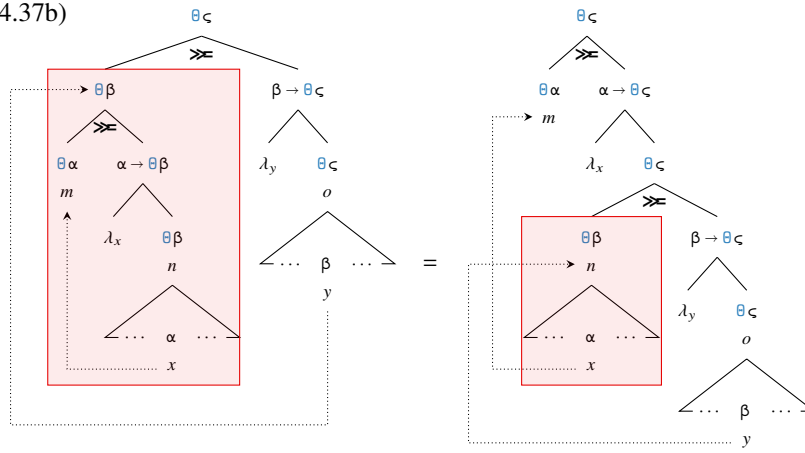
(4.36)



In fact, the exceptional scope of monadic effects can be seen directly in the **Associativity Law** of (4.16). This law is repeated in three different syntactic formats in (4.37). Imagine the constituent denoting n is an island. The derivations on the right-hand sides of (4.37) are island-violating. The m constituent is raised out of the n-island to scope over a larger chunk of the sentence, including o. But when Θ is a monad, these derivations are equivalent to those on the left-hand sides, which are island-respecting. The effectful constituent m takes scope over its enclosing island n, but nothing else. It is raised to the specifier of n, if you like. From there, the entire island is **pied-piped** to scope over the remainder of expression o. But the resulting meaning will be exactly as if m had moved alone.

(4.37a) $(m \gg\!= \lambda x.\, n\, x) \gg\!= \lambda y.\, o\, y \quad = \quad m \gg\!= (\lambda x.\, n\, x \gg\!= \lambda y.\, o\, y)$

(4.37b)



(4.37c)

```
do y <- do x <- m
           n x
   o y
```

=

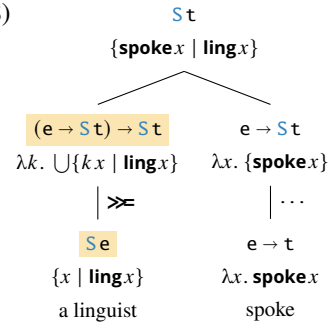
```
do x <- m
   y <- m x
   o y
```

4.3.2 Transformation into continuations

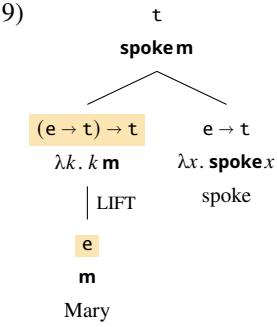
To recap the previous section, every monadic computation can be expressed in a `do`-block, structured as a sequence of right-nested actions, concluding with the calculation of an ordinary value. The resemblance to Quantifier Raising and traditional linguistic LFs is uncanny. It is natural then to wonder what monads have to do with Generalized Quantifiers. Following this thread exposes yet a third effect-driven derivational strategy.

To bring out the connection, let us treat (≫=) as a "type-shifting" unary operator, like η. In this form, it shifts an expression of type $\theta\,\alpha$ to one of type $(\alpha \to \theta\,\beta) \to \theta\,\beta$, as in (4.38).

(4.38)



(4.39)



In this guise, (≫=) plays a role very much like the traditional LIFT operator of Partee (1986). Where LIFT converts an ordinary value of type e into a Generalized Quantifier over properties of individuals, (≫=) converts an enriched value of type $\Theta\,\mathtt{e}$ into an enriched quantifier over properties that may have side effects.

In fact, the first monad law ensures that when a computation is trivial (just a value injected into some structure), then (≫=) is exactly equivalent to LIFT.

(4.40) **Left Identity:** $\eta\, x \gg\!= k = k\, x$

This is seen by simply rewriting the law in (4.40) using the unary version of (≫=).

(4.41) **Left Identity:** $(\eta\, x)^{\gg\!=} = \lambda k.\, k\, x$
$= \textsc{lift}\, x$

In tree form, this says the following two derivations must be equivalent:

(4.42)

$(\alpha \to \Theta\beta) \to \Theta\beta$
$\lambda k.\, \eta\, x \gg\!= k$
≫=
$\Theta\alpha$
$\eta\, x$
η
α
x

(4.43)

$(\alpha \to \Theta\beta) \to \Theta\beta$
$\lambda k.\, k\, x$
LIFT
α
x

And here's where things get interesting. As characterized in Chapter 2, Generalized Quantifiers themselves constitute a kind of computation. Any expression of type $(\alpha \to \mathtt{t}) \to \mathtt{t}$ can appear locally in a position where an ordinary α is expected. But of course the quantifier does not merely contribute a value to that position. Rather, it tests what happens when the rest of the derivation — its **continuation** — is run with different values of type α, and then makes a summary decision based on the results of these experiments. Here are the functorial and applicative instances for such continuized computations.

(4.44a) $k \bullet m := \lambda c.\, m\, (\lambda x.\, c\, (k\, x))$

(4.44b) $F \circledast X := \lambda c.\, F\, (\lambda f.\, X\, (\lambda x.\, c\, (f\, x)))$

From an algebraic perspective, there is nothing special about computations that expect their derivational contexts to compute *truth values*, per se; in other words, nothing special about the type `t`. Much more generally, any function of type $(\alpha \to \mathtt{o}) \to \rho$ can be construed as a computation that runs by swallowing some $\mathtt{o}$-sized chunk of its context. With this context it tries out different α values to see what $\mathtt{o}$ results they lead to. With the collection of these results in hand, it makes a decision on which ρ to return. When $\mathtt{o}$ and ρ are `t`, it (often) makes sense to think of these computations as "quantifications", in that the boolean that is returned may depend only on *how many* type-α values tipped the text toward truth. When $\mathtt{o}$ and ρ are other types, they may not do anything that we would associate with with quantities, but they are still functions of their contexts.

To the point, any higher-order function of type $(\alpha \to \Theta\,\beta) \to \Theta\,\beta$ fits this pattern. Such a function may well be construed as a computation purporting to be an α, but in reality expecting to see what *computations* $\Theta\,\beta$ result from filling in its local position with various type-α choices. And every unary application of $(\gg\!=)$ creates a higher-order function of exactly this type.

In light of this, we may choose to conceive of $(\gg\!=)$ not as a mode of combination, but as a way of transforming one kind of computation into another. Formally, this amounts to assigning $(\gg\!=)$ the type in (4.45), where $\mathtt{C}$ is understood to be the type of computations that depend on their continuations, in this case continuations of type $(\alpha \to \Theta\,\beta)$.

(4.45) $\quad (\gg\!=) :: \Theta\,\alpha \to \mathtt{C}\,\alpha$

Polymorphic functions like this, that convert values in one functor to values in another functor, are known as **natural transformations**.

Look at what happens when an arbitrary monadic computation $m :: \Theta\,\alpha$ is so transformed. The resulting computation $m^{\gg\!=} :: \mathtt{C}\,\alpha$ will have functorial and applicative combinators, induced by those of the $\mathtt{C}$ effect defined in (4.44):

(4.46a) $\quad k \bullet m^{\gg\!=} := \lambda c.\, m^{\gg\!=}\,(\lambda x.\, c\,(k\,x))$

(4.46b) $\quad F^{\gg\!=} \circledast X^{\gg\!=} := \lambda c.\, F^{\gg\!=}\,(\lambda f.\, X^{\gg\!=}\,(\lambda x.\, c\,(f\,x)))$

Strikingly, these induced operations are almost exactly identical to those in (4.19), which themselves were induced directly from the underlying monad Θ. The only difference is that where the original definitions in (4.19) "return" the underlying results with η, the continuized definitions in (4.46) leave open what will happen to them next by abstracting over this role with the continuation variable c. Naturally, passing η in for this function yields the original definitions. That is, for any monad Θ, we have the equivalences in (4.47). These equations

are made slightly more telegraphic by writing $(\cdot)^{\gg=}$ as $(\cdot)^{\uparrow}$ and $(\cdot)\,\eta$ as $(\cdot)^{\downarrow}$, to indicate lifting into and lowering out of the $\mathtt{C}$ effect space.

$$\text{(4.47a)}\quad k \bullet_{\mathtt{\Theta}} m = (k \bullet_{\mathtt{C}} m^{\gg=})\,\eta = (k \bullet_{\mathtt{C}} m^{\uparrow})^{\downarrow}$$

$$\text{(4.47b)}\quad F \circledast_{\mathtt{\Theta}} X = (F^{\gg=} \circledast_{\mathtt{C}} X^{\gg=})\,\eta = (F^{\uparrow} \circledast_{\mathtt{C}} X^{\uparrow})^{\downarrow}$$

This means that in principle, every single instance of $(\bullet)$ and $(\circledast)$, or the corresponding **R/L** and **A** meta-combinators, can be simulated with instances $(\bullet_{\mathtt{C}})$ and $(\circledast_{\mathtt{C}})$, provided access to free applications of the $(\cdot)^{\uparrow}$ and $(\cdot)^{\downarrow}$ operators above. In the framework laid out here, eliminating all effect-specific instances of $(\bullet)$ and $(\circledast)$ is as simple as restricting the **R**, **L**, and **A** rules to apply only to $\mathtt{C}$-type computations, rather than arbitrary $\mathtt{\Theta}$ computations. The relevant components of such a grammar are shown in Figure 9.

Going further, we might even replace the free *applications* of an expression m to η (provided by $(\cdot)^{\downarrow}$) with completely free *occurrences* of η in a derivation. That is, we might suppose that not only can expressions be freely applied to η, but η can also freely apply to expressions (as it was in derivations like (4.34), for instance). Together with $(\cdot)^{\uparrow}$, this would render the Unit and Join rules otiose. In fact, with free insertions of η, the **R** and **L** are also unnecessary. For an at-length presentation of compositional semantics in this style, see Charlow (2014).

4.4 Fused effects

Despite their prominent practical and theoretical roles in functional programming, monadic techniques for handling multiple effects face a well-known obstacle: the composition of two monads is not necessarily monadic. That is, there is no general way to define a law-abiding function $\mu_{\mathtt{\Theta K}} :: \mathtt{\Theta}\,(\mathtt{K}\,(\mathtt{\Theta}\,(\mathtt{K}\,\alpha))) \to \mathtt{\Theta}\,(\mathtt{K}\,\alpha)$, even when $\mathtt{\Theta}\,\alpha$ is a monad with an associated $\mu_{\mathtt{\Theta}} :: \mathtt{\Theta}\,(\mathtt{\Theta}\,\alpha) \to \mathtt{\Theta}\,\alpha$ and $\mathtt{K}\,\alpha$ is a monad with $\mu_{\mathtt{K}} :: \mathtt{K}\,(\mathtt{K}\,\alpha) \to \mathtt{K}\,\alpha$. The reason, intuitively, is that $\mathtt{\Theta}$ and $\mathtt{K}$ are interleaved in $\mu_{\mathtt{\Theta K}}$, which means there's no way to use the respective $\mu_{\mathtt{\Theta}}$ and $\mu_{\mathtt{K}}$ to help flatten out the layers of effects.

To see a situation where this fact might rear its head, consider the description in (4.49). For the purposes of the demonstration, imagine that 'another' is a determiner with both anaphoric and indeterminate effects. It requires an antecedent — *other than what?* — and generates a set of possible referents. Which referents it computes depends on the antecedent: roughly those left in

Types:

$\vdots$

Combinators:

$\vdots$

Type-shifters:

$(\cdot)^{\uparrow} :: (\Theta\,\alpha) \to \mathsf{C}\,\alpha$ Up

$m^{\uparrow} \coloneqq \lambda k.\, k \star m$

$(\cdot)^{\downarrow} :: (\mathsf{C}\,\alpha) \to \Theta\,\alpha$ Down

$m^{\downarrow} \coloneqq m\,\eta$

Meta-combinators:

$\mathbf{L} :: (\sigma \to \tau \to \omega) \to \mathsf{C}\,\sigma \to \tau \to \mathsf{C}\,\omega$ Map Left

$\mathbf{L}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\, a * E_2) \bullet E_1$

$\mathbf{R} :: (\sigma \to \tau \to \omega) \to \sigma \to \mathsf{C}\,\tau \to \mathsf{C}\,\omega$ Map Right

$\mathbf{R}\,(*)\,E_1\,E_2 \coloneqq (\lambda b.\, E_1 * b) \bullet E_2$

$\mathbf{A} :: (\sigma \to \tau \to \omega) \to \mathsf{C}\,\sigma \to \mathsf{C}\,\tau \to \mathsf{C}\,\omega$ Structured App

$\mathbf{A}\,(*)\,E_1\,E_2 \coloneqq (\lambda a \lambda b.\, a * b) \bullet E_1 \circledast E_2$

$\vdots$

Figure 9 A monadic grammar whose binary rules use continuations

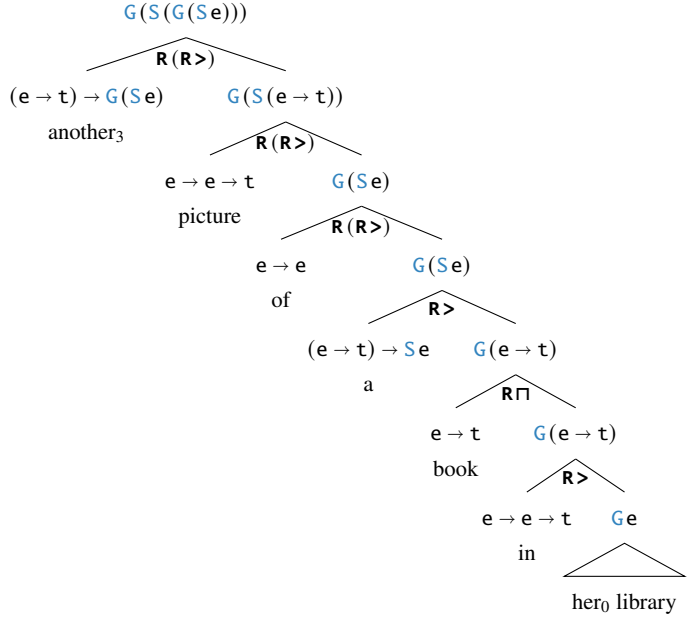
its restrictor once the antecedent is removed. Its type, therefore, is plausibly rendered as $(\mathtt{e} \to \mathtt{t}) \to \mathtt{G}(\mathtt{S}\,\mathtt{e})$, with the lexical semantics in (4.48).

(4.48) $\text{another}_i :: (\mathtt{e} \to \mathtt{t}) \to \mathtt{G}(\mathtt{S}\,\mathtt{e})$

$[\![\text{another}_i]\!] \coloneqq \lambda P \lambda g.\, \{x \mid Px,\ x \neq g_i\}$

Putting this determiner together with a restrictor that itself contains both anaphoric and indeterminate effects looks as in (4.49).

(4.49)



There is no other way to combine these pieces, given the grammar in Figure 8. This is a shame, since there's no *semantic* reason why the assignment-dependencies induced by 'another' and 'her library' cannot be collapsed into a single G-layer. Likewise the alternatives generated by 'a' and those by 'another' could certainly in principle be represented in a single set determined by the cross product of the two restrictors. Indeed, the meaning given in (4.50) is a perfectly plausible denotation for the phrase, in line with the sorts of denotations otherwise assigned to nested indefinite expressions here. The trouble is just that there's no way to compose the determiner and restrictor using the **J** rules based on G and S that would bring it about.

(4.50) $(4.49) :: \mathtt{G}(\mathtt{S}\,\mathtt{e})$

$[\![(4.49)]\!] \coloneqq \lambda g.\, \{x \mid (\mathbf{book} \sqcap \mathbf{in}\,(\mathbf{lib}\,g_0))\,y,\ \mathbf{pic}\,y\,x,\ x \neq g_3\}$

And here's the kicker: the type $\mathtt{G}(\mathtt{S}\,\alpha) = \mathtt{r} \to \{\alpha\}$ *is a monad*! Here's its μ:

(4.51) $\mu_{\mathtt{GS}} :: \mathtt{G}(\mathtt{S}\,(\mathtt{G}\,(\mathtt{S}\,\alpha))) \to \mathtt{G}(\mathtt{S}\,\alpha)$

$\mu_{\mathtt{GS}}\,M \coloneqq \lambda g.\, \bigcup\{m\,g \mid m \in M\,g\}$

The trouble again is that this is an entangled mixture of the μs of G and S, rather than a composition of those functions. In other words, this μ treats GS as a single sort of environment-sensitive computation with parallel outputs, rather than a combination of independent environment-sensitivity and indeterminacy. We might, therefore, give it its own type constructor, say $\mathtt{H}\,\alpha ::= \mathtt{r} \to \{\alpha\}$, since this is essentially the type Hamblin (1973) works with in his seminal semantics on questions. The type of 'another' would then be $(\mathtt{e} \to \mathtt{t}) \to \mathtt{H}\,\mathtt{e}$ with its denotation unchanged. And with this conjoint effect bottled up in the effect signature, we could continue to use the type-driven apparatus we have developed so far to dispatch the right modes of combination wherever it is possible to do so.

Still, it's hard to deny that this is a blow to the modularity we have championed in this Element. For starters, in order to compose the phrase in (4.49), the semantics of 'a' and 'her' must be lifted to live in the H type space.

(4.52) $\mathrm{a} :: (\mathtt{e} \to \mathtt{t}) \to \mathtt{H}\,\mathtt{e}$

$[\![\mathrm{a}]\!] \coloneqq \lambda P \lambda g.\, \{x \mid P\,x\}$

(4.53) $\mathrm{her}_n :: \mathtt{H}\,\mathtt{e}$

$[\![\mathrm{her}_n]\!] \coloneqq \lambda g.\, \{g_n\}$

This is exactly the sort of generalization to the worst case that we have so far avoided. Yet it is possible that some combinations of intuitively separable effects nevertheless act together in a self-contained computational system deserving of its own encapsulated type. In the next section we give an example of such a system, building on the GS monad above. Then in Chapter 5, we show how to accomplish many of the same empirical goals while dissolving the system back into its independent components.

4.4.1 Monad Transformers

Take another look at (4.51). Notice that we can rewrite this definition using the underlying monadic combinators of S:

(4.54) $\mu_{\mathtt{GS}}\, M \coloneqq \lambda g.\ (\lambda m.\ m\, g) \star_{\mathtt{S}} M\, g$

This is no accident. For *any* monad $\mathtt{\Theta}$, replacing $(\star_{\mathtt{S}})$ with $(\star_{\mathtt{\Theta}})$ in (4.54) would deliver a well-typed $\mu_{\mathtt{G\Theta}}$ for the composite effect signature $\mathtt{G\,\Theta}$. Given this, we might define a **higher-order constructor** $\mathtt{G}^{+}$ as in (4.55), parameterized by an inner constructor $\mathtt{\Theta}$ as well as a concrete type α.

(4.55) $\mathtt{G}^{+}\,\mathtt{\Theta}\,\alpha ::= \mathtt{r} \to \mathtt{\Theta}\,\alpha$

Where an ordinary constructor maps types to types, a higher-order constructor like the one in (4.55) maps constructors to constructors. They take kinds of computations as arguments and produce enhanced computations as results. From this perspective, $\mathtt{G}^{+}$ *adds* the ability to read and respond to an environment to an existing computation $\mathtt{\Theta}$.

And it's not hard to prove that whenever the inner computation $\mathtt{\Theta}$ is monadic, this enhanced environment-sensitive computation $\mathtt{G}^{+}\mathtt{\Theta}$ will also satisfy the monad laws. Higher-order constructors like this, that produce new monads from old ones, are sometimes known as **monad transformers** (Liang, Hudak, & Jones, 1995).

Notice also that the basic $\mathtt{G}$ monad is exactly the $\mathtt{G}^{+}$ transformer applied to the **identity monad**, defined in (4.56).

(4.56) $\mathtt{I}\,\alpha ::= \alpha$

$k \bullet_{\mathtt{I}} m \coloneqq k\, m$

$\mu_{\mathtt{I}}\, M \coloneqq M$

The $\mathtt{I}$ constructor represents a computation that doesn't do anything but hold a value. Mapping a function over an $\mathtt{I}$ computation is just function application, since an $\mathtt{I}$ computation is nothing but an argument. Using the definitions in (4.56), we see that $(\star_{\mathtt{I}})$ is also just function application. So the equation in (4.54) with $(\star_{\mathtt{I}})$ in place of $(\star_{\mathtt{S}})$ just reduces to **W**, i.e., $\mu_{\mathtt{G}}$.

It turns out, almost all of the effects in Table 2 can be seen as applications of some transformer to the identity monad. That is, they are all special cases of the act of *enhancing* an existing computation with a new, particular effect; namely, the special case when that enhancement is preformed on the trivial computation $\mathtt{I}$ that merely holds a value.

We will not dwell on the applications of this technique (see Shan 2001a), except to draw attention to one interesting case, that of $\mathtt{T}$. The transformer that gives rise to $\mathtt{T}$ when applied to the identity monad is given in (4.57). This constructor enhances a computation $\mathtt{\Theta}$ by adding the ability to read and write to a common state $\mathtt{s}$ that is carried throughout the computation.

(4.57) $\mathtt{T}^{+}\,\Theta\,\alpha ::= \mathtt{s} \to \Theta\,(\alpha \times \mathtt{s})$

$k \bullet_{\mathtt{T}^{+}\Theta} m := \lambda i.\, k \bullet_{\Theta} m\, i$

$\mu_{\mathtt{T}^{+}\Theta}\, M := \lambda i.\, (\lambda\langle m,\, j\rangle.\, m\, j) \star_{\Theta} M\, i$

When we apply this transformer to S, we get $\mathtt{T}^{+}\,\mathtt{S} \equiv \mathtt{D}$, the type representing computations in the style of dynamic semantics. In typical presentations of dynamic semantics, pronouns, discourse, referents, and indefinites interact in a single pervasive framework encompassing the interleaved effects of reading, writing, and nondeterminism. For instance, we might find lexical entries like those in (4.58).

(4.58a) $\text{it}_n :: \mathtt{D\,e}$

$[\![\text{it}_n]\!] := \lambda i.\, \{\langle i_n,\, i\rangle\}$

(4.58b) $\triangleright :: \mathtt{e} \to \mathtt{D\,e}$

$[\![\triangleright]\!] := \lambda x \lambda i.\, \{\langle x,\, x +\!\!+ i\rangle\}$

(4.58c) a planet $:: \mathtt{D\,e}$

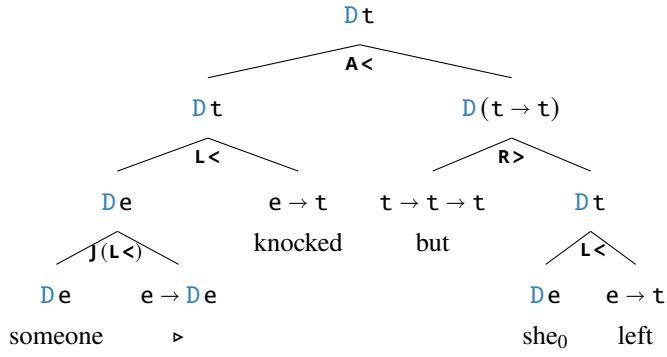
$[\![\text{a planet}]\!] := \lambda i.\, \{\langle x,\, i\rangle \mid \textbf{planet}\, x\}$

(4.58d) $\text{if} :: \mathtt{D\,t} \to \mathtt{D\,t} \to \mathtt{D\,t}$

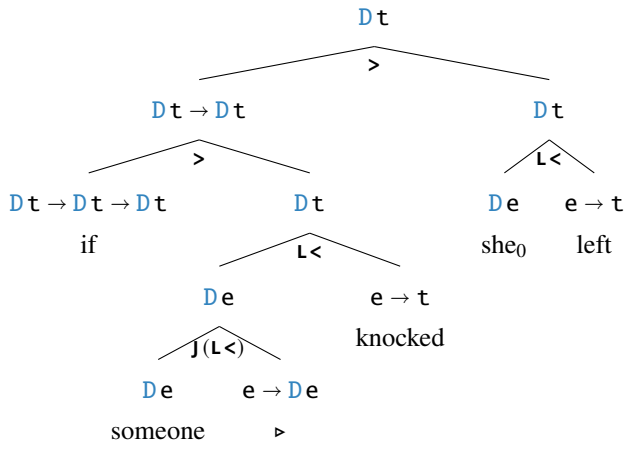
$[\![\text{if}]\!] := \lambda m \lambda n \lambda i.\, \{\langle \forall \langle p,\, j\rangle \in m\, i.\, p \Rightarrow \exists \langle q,\, k\rangle \in n\, j.\, q,\, i\rangle\}$

And with these, we might put together derivations of such classic dynamic phenomena as cross-sentential and donkey binding, as in (4.59) and (4.60).

(4.59)



(4.60)



As elegant and well-studied as this system is, one can hardly shake the feeling that it is overcooked. Nothing brings this out more than the canonical lexical entries in (4.58a)–(4.58c). The pronoun doesn't do anything interesting except read from the input i; it is deterministic and doesn't change the state i at all. The referent-pushing operator ▷ adds its prejacent to the state, but is otherwise boring; it is deterministic and doesn't read from its input. And the indefinite ramifies the state, adding a new thread for each witness, but does not on its own interact with the state in any way. Yet the three are typed uniformly as $\mathtt{D}\,\mathtt{e}$.

In the Chapter 5, we add one last mode of combination based on Category Theoretic notions, and show that this conflation can be avoided.

4.5 Implementing monadic effects in the type-driven interpreter

Extending the interpreter with a monadic mode of combination requires the same, now familiar modifications as in the last two chapters. We first add a mode `JN` representing the meta-mode **J**.

```
data Mode
  = FA | BA | PM       -- etc
  | MR Mode | ML Mode  -- map right and map left
  | AP Mode            -- structured app
  | UR Mode | UL Mode  -- unit right and unit left
  | JN Mode            -- join
```

To determine the applicability of the `JN` rule, we'll need to declare which effects are monadic. Again, as it happens, all of the applicative effects presented in this chapter are also monadic, so this predicate is extensionally equivalent to `applicative`. Still, we include it for conceptual hygiene so that the code reads in a manner responsive to the truth.

```
functor, applicative, monad :: EffX -> Bool
functor     _      = True
applicative (WX s) = monoid s
applicative f      = functor f && True
monad f            = applicative f && True

monoid :: Ty -> Bool
monoid T = True
monoid _ = False
```

However, the logic determining when to dispatch `JN` must be slightly different than that of the others. To see this, compare the types of the **R** and **J** modes, repeated below.

(4.61) $\mathbf{R}\,(* :: \sigma \to \tau \to \upsilon) :: \sigma \to \Theta\,\tau \to \Theta\,\upsilon$

(4.62) $\mathbf{J}\,(* :: \sigma \to \tau \to \Theta\,(\Theta\,\upsilon)) :: \sigma \to \tau \to \Theta\,\upsilon$

Provided with a function $(*) :: \sigma \to \tau \to \upsilon$, the **R** rule returns a mode of combination of type $\sigma \to \Theta\,\tau \to \Theta\,\upsilon$. This means that $\mathbf{R}\,(*)$ is only ever applicable when the right daughter is of type $\Theta\,\tau$. Hence the logic of `addMR`:

```
addMR l r = case r of
  Comp f t | functor f
    -> [ (MR op, Comp f u) | (op, u) <- combine l t ]
  _ -> [                                            ]
```

This rule only fires when the right daughter's type has a particular shape: `Comp f t`. Otherwise it immediately return an empty list. And when that daughter does in fact denote a computation, `combine` is called recursively on the *underlying* type `t`. Because a type is finite, this strategy is guaranteed to terminate. At every step it strips off some effect wrapper and tries again with what is left.

But the **J** meta-combinator is different. Provided with a function $(*) :: \sigma \to \tau \to \Theta\,(\Theta\,\upsilon)$, it returns a mode of combination of type $\sigma \to \tau \to \Theta\,\upsilon$. This means, in principle, that it could apply to *any* two types σ and τ, if there happens

to be a way to combine them to yield something of type `θ(θυ)`. There is thus no way to know simply by looking at the shapes of the daughters' types `l` and `r` whether the **J** mode might apply.

Instead, we need to try and combine the daughters first, and then inspect the *results* to see if we ended up with anything joinable. Thus we split `combine` into two parts. The `binaryCombs` list holds all of the results computed in the preceding chapters; that is, all the up-front binary combinations we can find. Then the `unaryCombs` function sets to work on each result. At this point there are only two things to do. First, keep it! A good combination is still a good combination. Additionally, if its type happens to be `Comp f (Comp g a)`, where `f` and `g` are the same monadic effect, then go ahead and join it. Again, these are not exclusive. We hold on to both the layered and the lowered combinations, since they are both valid.

```
combine :: Ty -> Ty -> [(Mode, Ty)]
combine l r = binaryCombs >>= unaryCombs
  where
    binaryCombs =
      modes l r
      ++ addMR l r
      ++ addML l r
      ++ addAP l r
      ++ addUR l r
      ++ addUL l r
    unaryCombs e =
      -- keep any result from above
      return e
      -- and if it happens to have a two-layered
      -- monadic type, also join it
      ++ addJN e

addJN e = case e of
  (op, Comp f (Comp g a)) | f == g, monad f
    -> [ (JN op, Comp f a) ]
  _ -> [                   ]
```

Note that `binaryCombs` is a list of modes. And `unaryCombs` is a function from modes to an extended list of modes. Fittingly, the way to take each element of the former, pass it in to the latter, and then flatten out all the results, is just the `>>=` operation on lists. In programs, as in language, this pattern just has a way of showing up.

5 Eliminating effects

5.1 Adjunctions

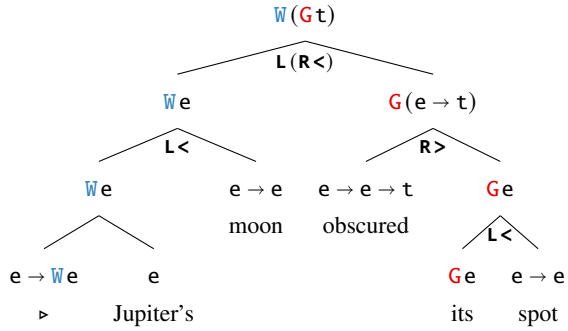
Let us return to the issue of anaphora. Intuitively, we would like to analyze a pronoun as a computation that retrieves a salient discourse referent from memory. In its simplest form, something like (5.1a). And we would like to analyze its antecedent as a computation that stores a discourse referent in memory. Again with maximal simplification, something like the operator in (5.1b) might suffice.

(5.1a) $\mathrm{it} :: \mathtt{G}\,\mathtt{e}$
$[\![\mathrm{it}]\!] := \lambda x.\, x$

(5.1b) $\triangleright :: \mathtt{e} \to \mathtt{W}\,\mathtt{e}$
$[\![\triangleright]\!] := \lambda x.\, \langle x, x \rangle$

In this construal, the pronoun and its antecedent constitute semantically and type-theoretically distinct effects. This alone is no barrier to composition, given any of the type-driven grammars in the preceding chapters. But in all of those grammars, the result of putting together a sentence with both an antecedent and a pronoun will be a computation with two effects. The antecedent will survive in memory, and the pronoun will continue to await its resolution, as in (5.2).

(5.2)



In other words, we have reading, and we have writing, but we don't have **binding**. In Chapters 3 and 4, we sketched solutions to this that are typical of dynamic approaches to semantics. These analyses create a new, generalized effect that models any sort of interaction with a **discourse state**. In general, a sentence with unresolved pronouns and fresh new antecedents will denote a

state transition that both reads from and modifies the state. But on their own, pronouns will comprise the special case of programs that read from but do not modify the state, and antecedents the special case of programs that modify but do not inspect the state. In this sense, as suggested in Chapter 4.4.1, the dynamic solution is a generalization to the worst case.

Here, we offer a different solution, inspired by Shan (2001b). The essential intuition is that a discourse in which all pronouns are bound should denote an ordinary proposition, a **pure** value. Once the antecedent has supplied its referent to the pronoun, both the writing and the reading effects should be considered resolved. The two effects are closure operators to one another.

This duality is mathematically manifest in the isomorphism defined by the following converse functions.

$$\begin{aligned} &(5.3) \quad \phi :: (\alpha \to \mathsf{G}\beta) \to \mathsf{W}\,\alpha \to \beta && \psi :: (\mathsf{W}\,\alpha \to \beta) \to \alpha \to \mathsf{G}\beta \\ &\qquad \phi := \lambda k \lambda \langle a,\, g \rangle .\, k\, a\, g && \psi := \lambda c \lambda a \lambda g .\, c\, \langle a,\, g \rangle \end{aligned}$$

Any **Kleisli arrow** into a G-computation can be converted to a **co-Kleisli arrow** from a W-computation by the function ϕ. And any co-Kleisli arrow from W can be converted to a Kleisli arrow into G. And these transformations proceed without loss of information. They are invertible. That is, $\psi\,(\phi\,k) = k$ for any $k :: \alpha \to \mathsf{G}\beta$, and $\phi\,(\psi\,c) = c$ for any $c :: \mathsf{W}\,\alpha \to \beta$. Setting aside the type constructors, this isomorphism is just the familiar equivalence between curried and uncurried presentations of multi-argument functions.

Whenever two functors Ω and Γ have this dual property, they are said to be **adjoint**. Specifically, Ω is **left adjoint** to Γ, written $\Omega \dashv \Gamma$, when functions *into* Ω are isomorphic to functions *from* Γ. In the case at hand, we say: $\mathsf{W} \dashv \mathsf{G}$.

Every adjunction $\Omega \dashv \Gamma$ gives rise to a pair of functions, ε and η, by applying the components of this isomorphism to identity functions. These functions are known respectively as the **co-unit** and **unit** of the adjunction. For the $\mathsf{W} \dashv \mathsf{G}$ adjunction specified in (5.3), this yields the functions (5.4).

$$\begin{aligned} &(5.4) \quad \varepsilon :: \mathsf{W}\,(\mathsf{G}\,\alpha) \to \alpha && \eta :: \alpha \to \mathsf{G}\,(\mathsf{W}\,\alpha) \\ &\qquad \varepsilon := \phi\,\mathbf{id} && \eta := \psi\,\mathbf{id} \\ &\qquad\quad = \lambda \langle f,\, g \rangle .\, f\, g && \quad = \lambda a \lambda g .\, \langle a,\, g \rangle \end{aligned}$$

It turns out that whenever $\Omega \dashv \Gamma$ is an adjunction, the composite functor $\Gamma\,(\Omega\,\alpha)$ is a monad, and the η function determined by (5.4) is in fact its η. This guarantees that there is always a way of lifting any ordinary value $x :: \alpha$ into a trivial computation $m :: \Omega\,(\Gamma\,\alpha)$ that has the structure required of Ω and Γ.

The ε function, on the other hand, is entirely new. It guarantees that any composite computation of type $\Gamma\,(\Omega\,\alpha)$ can be *deconstructed*, eliminating all

of the Γ and Ω structure. How does this work in the case of $\mathbb{W} \dashv \mathbb{G}$? Well, a computation of type $\mathbb{W}(\mathbb{G}\alpha) ::= \mathtt{r} \times (\mathtt{r} \to \alpha)$ is a pair consisting of two things: an environment $g :: \mathtt{r}$ and a function from environments to values $f :: \mathtt{r} \to \alpha$. To extract that ordinary value of type α, we need only to pass the stored context g into the context-dependent function f.

5.1.1 Adjunction as a higher-order mode of combination

Note that adjunction is a binary, asymmetric relation between functors. In this respect, constructing a mode of combination that takes advantage of adjunctions between effects is particularly straightforward. After all, a mode of combination is itself a binary, asymmetric relation between denotations. We could, for instance, imagine the forward and backward co-unit combinators in (5.5).

(5.5a) $(\blacktriangleleft_{>}) :: \Omega(\alpha \to \beta) \to \Gamma\alpha \to \beta$

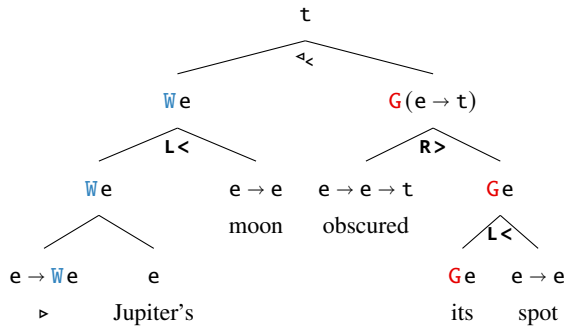
$L \blacktriangleleft_{>} R := \varepsilon((\lambda f.\ (\lambda x.\ f\,x) \bullet R) \bullet L)$

(5.5b) $(\blacktriangleleft_{<}) :: \Omega\alpha \to \Gamma(\alpha \to \beta) \to \beta$

$L \blacktriangleleft_{<} R := \varepsilon((\lambda x.\ (\lambda f.\ f\,x) \bullet R) \bullet L)$

These might be used in a derivation like (5.6), which is identical to the earlier (5.2) except for the last step, where this time binding is achieved. Notice that this is a kind of dynamic or discourse binding; the referent persists *up* the computation of the left branch, until the subject meets the predicate, at which point it sinks *down* the right branch to value the pronoun.

(5.6)



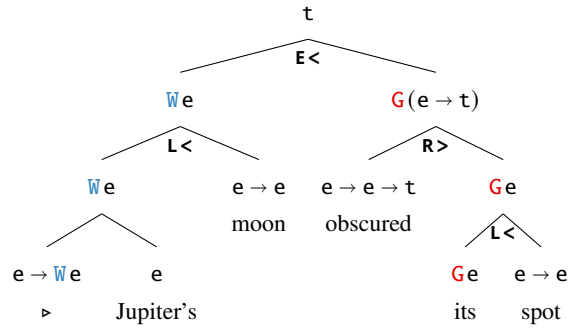
But as usual, this first-order combinatorial approach would need special cases for every basic mode of combination (in addition to > and <), and then would inevitably fall flat in the presence of other, irrelevant effects. So we generalize

in the typical way: whenever there is any way of combining the underlying types of a Ω constituent and a Γ constituent, where $\Omega \dashv \Gamma$, we can combine the constituents by mapping over the two effects, combining the underlying values, and then applying ε to the result. The complete type-driven grammar, extending that of the last chapter, is given in Figure 10.

(5.7) $\quad \mathbf{E}\,(*)\,E_1\,E_2 \coloneqq \varepsilon\,((\lambda l.\,(\lambda r.\,l * r) \bullet E_2) \bullet E_1)$

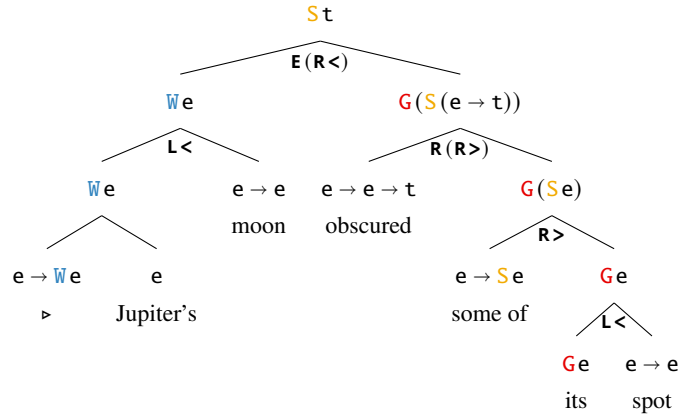
The simple combinators $(\triangleleft_{>})$ and $(\triangleleft_{<})$ are thus reproduced as **E>** and **E<**. And the derivation in (5.6) is equivalent to the one in (5.8).

(5.8)



What's more, incidental effects above and below the adjoint ones no longer interfere with the adjunction, as they shouldn't. For instance, adding some indeterminacy in one of the constituents of (5.8), as in (5.9), does not preclude binding.

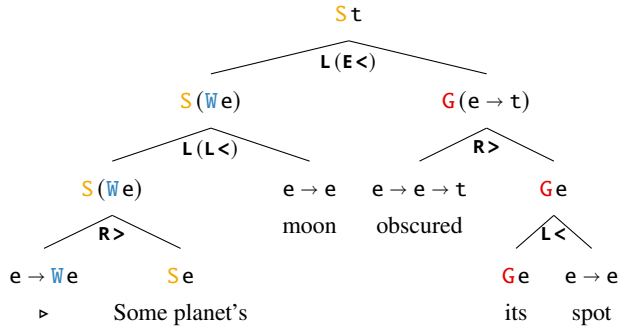
(5.9)



The first-order combinators ($\blacktriangleleft_{>}$) and ($\blacktriangleleft_{<}$) would not have known what to do in this circumstance, as the underlying types — $\mathtt{e}$ and $\mathtt{S}(\mathtt{e} \to \mathtt{t})$ — cannot be combined by any basic mode of combination. But with $\mathbf{E}$, these underlying types are combined in the obvious way — with $\mathbf{R}<$ — and then the $\mathtt{W}$ and $\mathtt{G}$ effects cancel each other out. The referent coming from the left is passed into the function requesting a referent on the right, and only the indeterminacy remains.

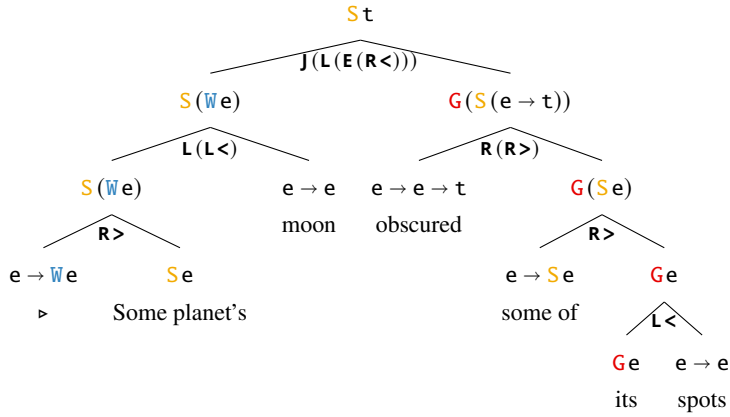
The example in (5.9) demonstrates how binding into an indefinite emerges. The inverse is also possible.

(5.10)



And for the *coup de grâce*: indefinites can also bind into other indefinites. This is shown in (5.11).

(5.11)



Let us walk through the final step. The $\mathbf{J}$ meta-combinator from the previous chapter combines the two daughters via the mode of combination determined by its argument — $\mathbf{L}(\mathbf{E}(\mathbf{R}<))$ — before flattening the result with μ. That inner

Types:

$\tau ::= \mathtt{e} \mid \mathtt{t} \mid \ldots$ Primitive types
$\mid \tau \to \tau$ Function types
$\mid \ldots$ ...
$\mid \Sigma\tau$ Computation types

Effects:

$\Sigma ::= \mathtt{G}$ Reading
$\mid \mathtt{S}$ Indeterminacy
$\mid \ldots$ ...

Basic Combinators:

$(\mathbf{>}) :: (\alpha \to \beta) \to \alpha \to \beta$ Forward Application
$f \mathbf{>} x \coloneqq f\,x$

$(\mathbf{<}) :: \alpha \to (\alpha \to \beta) \to \beta$ Backward Application
$x \mathbf{<} f \coloneqq f\,x$

...

Meta-combinators:

$\mathbf{L} :: (\sigma \to \tau \to \upsilon) \to \Theta\sigma \to \tau \to \Theta\upsilon$ Map Left
$\mathbf{L}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\ a * E_2) \bullet E_1$

$\mathbf{R} :: (\sigma \to \tau \to \upsilon) \to \sigma \to \Theta\tau \to \Theta\upsilon$ Map Right
$\mathbf{R}\,(*)\,E_1\,E_2 \coloneqq (\lambda b.\ E_1 * b) \bullet E_2$

$\mathbf{A} :: (\sigma \to \tau \to \upsilon) \to \Theta\sigma \to \Theta\tau \to \Theta\upsilon$ Structured App
$\mathbf{A}\,(*)\,E_1\,E_2 \coloneqq (\lambda a \lambda b.\ a * b) \bullet E_1 \circledast E_2$

$\acute{\mathbf{U}} :: ((\sigma \to \sigma') \to \tau \to \upsilon) \to (\Theta\sigma \to \sigma') \to \tau \to \upsilon$ Unit Right
$\acute{\mathbf{U}}\,(*)\,E_1\,E_2 \coloneqq (\lambda a.\ E_1\,(\eta\,a)) * E_2$

$\grave{\mathbf{U}} :: (\sigma \to (\tau \to \tau') \to \upsilon) \to \sigma \to (\Theta\tau \to \tau') \to \upsilon$ Unit Left
$\grave{\mathbf{U}}\,(*)\,E_1\,E_2 \coloneqq E_1 * (\lambda b.\ E_2\,(\eta\,b))$

$\mathbf{J} :: (\sigma \to \tau \to \Theta(\Theta\upsilon)) \to \sigma \to \tau \to \Theta\upsilon$ Join
$\mathbf{J}\,(*)\,E_1\,E_2 \coloneqq \mu\,(E_1 * E_2)$

$\mathbf{E} :: (\sigma \to \tau \to \upsilon) \to \Omega\sigma \to \Gamma\tau \to \upsilon$ Co-unit
$\mathbf{E}\,(*)\,E_1\,E_2 \coloneqq \varepsilon\,((\lambda a.\ (\lambda b.\ a * b) \bullet E_2) \bullet E_1)$

Figure 10 A type-driven grammar with adjunctions

complex combinator is a combination of the two top-level combinators in (5.10) and (5.9). As in (5.10), the outer **L** means we begin by skipping over the left daughter's top effect, S. This leaves us with W e on the left and G(S(e → t)) on the right. These are combined via **E**(**R**<), exactly as in (5.9). The result of these combined combinations, just before the μ imposed by the **J**, is of type S(S e). The outer S corresponds to the left indefinite, which was mapped over first, and the inner S to the right indefinite, which is mapped over while executing the ε of **E**. Finally, this doubly-layered set is unioned by the outermost **J**, delivering a single indeterminate proposition with witnesses varying by both planets and their spots.

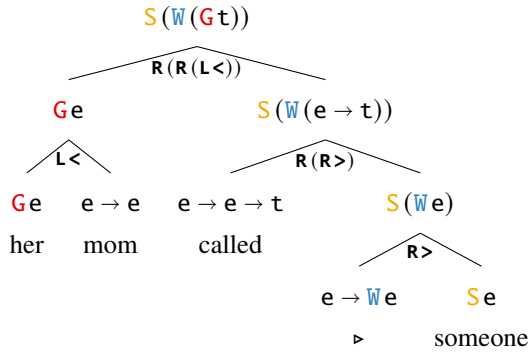
Note that this is exactly the sort of discourse binding made available by full-throttle dynamic semantic frameworks, like that of Chapter 4.4.1. There, the context-sensitivity, memory, and parallelism that characterize such frameworks were entangled in a single effect constructor D α ::= s → {α × s}. As a result, any expression that engaged in any one of these computational effects had to be typed as if it engaged in all of them, and its denotation had to include trivial structures — singleton sets, or unchanged or unused inputs — for those effects that were beside its point.

But in (5.11), there is no need to lift or reanalyze the individual effectful components. Indefinite denotations are indeterminate. Subjects get remembered. Pronouns need antecedents. These are the fundamental semantic properties of such expressions, and in (5.11), this is all there is to their denotations.

5.1.2 Crossover

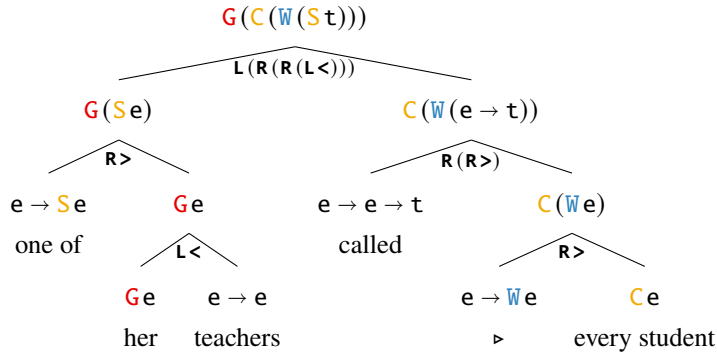
One fortuitous consequence of this formulation of binding is that it inherits the non-commutativity of adjunction. The **E** rule expects the left adjoint to come from the left daughter. For the W ⊣ G adjunction, this means that discourse antecedents must precede the pronouns they bind. Expressions in which this order is reversed are still composable, and even composable in such a way that the would-be binder outscopes its would-be bindee, but the two effects will never cancel out. Nothing in the grammar will ever pass the remembered referent coming from the right into the request for a referent coming from the left.

(5.12)



In this manner, the mode of combination derives the scope and binding pattern long known to linguists as **crossover**. Namely, discourse antecedence and retrieval proceeds from left to right, even while general semantic scope may be arbitrarily inverted. To put a point on this, consider the derivation in (5.13). The quantificational object takes logical scope over the indefinite subject, so that teachers vary with students. But since the computation still includes a G constructor, the pronoun's request for an antecedent remains open.

(5.13)



To be sure, there are many ways to combine the subject and predicate of (5.13), which determine different layerings of the various effects. But none of them will fill the pronoun's request for an antecedent. The best that one can do is scope the object's referent W over the subject's request G, yielding the layering C(W(G(St))). But as there is no independent ε-mechanism beside the **E** rule, they will just live on like this, anaphoric ships passing in the night.[3]

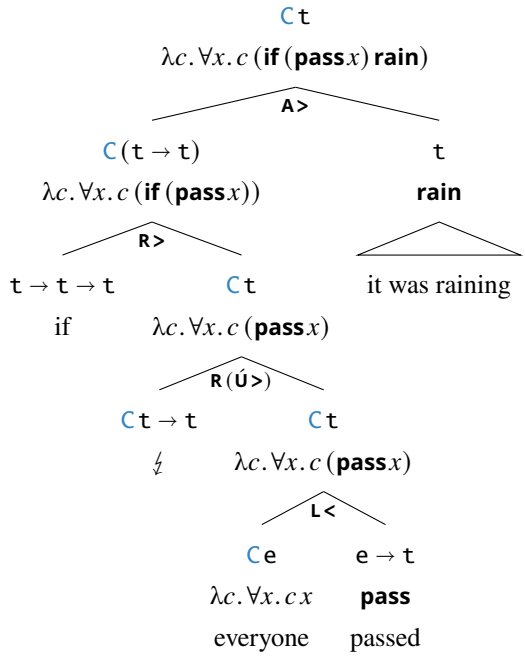
[3]See Barker and Shan (2014) for a crossover solution with a similar character. Barker and

5.2 Islands

Some effects may be delimited not by any particular closure operator, but rather by a certain syntactic configuration. Loosely following common parlance in linguistic literature, we will refer to such encapsulating domains as **islands**. For example, the scopes of distributive quantifiers like 'every' and 'no' are almost always bounded by their enclosing finite clauses, regardless of what other expressions they co-occur with.

It is tempting to associate such scope-delimiting nodes with obligatory closure operators, but a moment's reflection on the discussion in Chapter 3.3 will show that this will not in general suffice. There it is already demonstrated how nondeterminism can spill right over the edge of an existential closure operator, permitting the sort of exceptional scope associated with indefinites. With distributive quantifiers, the story is no different. The troublesome derivation would look as in (5.14).

(5.14)



Shan manage all effect combinations using layers of continuations, as sketched in Chapter 4.4.1. Rightward referent introductions may outscope leftward pronouns, but the two continuation layers can never be merged. In contrast, leftward referents and rightward pronouns may simply meet on the same level, where they cancel each other out.

Let's say the closure operator $\lightning$ has the Lowering semantics assigned by Barker and Shan (2014) in (5.15).

(5.15) $\lightning :: \mathtt{C}\,\mathtt{t} \to \mathtt{t}$

$\lightning := \lambda m.\, m\,\mathbf{id}$

If it were to apply directly to the conditional antecedent, it would result in the ordinary proposition true if everyone passed, false if anyone didn't.

(5.16) $\lightning [\![\text{everyone passed}]\!] = [\![\text{everyone passed}]\!]\,\mathbf{id}$

$= (\lambda c.\, \forall x.\, c\,(\mathbf{pass}\,x))\,\mathbf{id}$

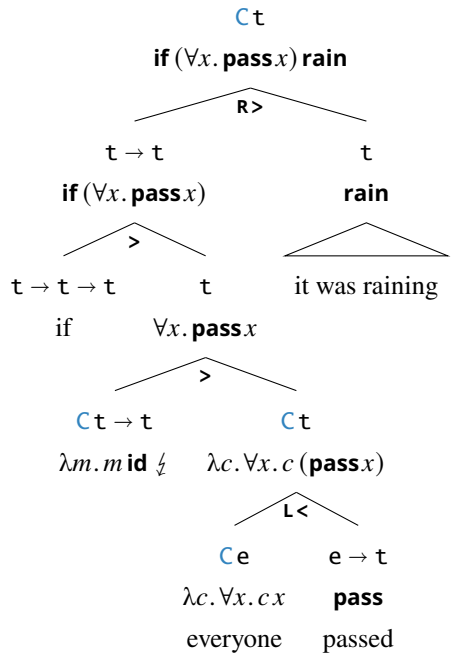
$= \forall x.\, \mathbf{pass}\,x$

But in (5.14), the operator does not apply directly to the antecedent. Instead it is mapped over the antecedent by $\mathbf{R}$, and then applied to a η-ified version of the underlying proposition. That underlying proposition is everything in the scope of the continuation, roughly $\mathbf{pass}\,x$. Applying η to this results in $\lambda k.\, k\,(\mathbf{pass}\,x)$. And then applying $\lightning$ to this LIFTed proposition brings us right back where we started: $\mathbf{pass}\,x$. The net result is a no-op, and the antecedent remains a quantificationally-charged computation. The conditional operator is mapped over it, and the island is escaped.

How then can narrow scope be enforced? One of the benefits of the effect-theoretic approach we have taken is that the types of nodes are revealing of their contents. So far we have used this information only to determine the ways that two constituents may be combined. But we might just as easily use the information to curtail, or boost, certain kinds of interpretations. Looking again at the tree in (5.14), we can tell at a glance that something has gone empirically awry because the restrictor has unclosed continuations in it. That is, its type includes the letter C. That's enough to know that some quantifier has not yet closed its scope, and if nothing is done, interpretations will be generated in which it continues to gobble up, and quantify over, more of its syntactic context.

So one way to implement quantificational scope islands is simply to filter out any derivations with C-effects in their types. In other words, islandhood may be type-driven. Assuming tensed clauses constitute such an island for quantifiers, the derivation in (5.14) would never arise because the conditional antecedent does not have a valid type for a tensed clause. Fortunately, the types do allow for other derivations, like the obvious (5.17).

(5.17)



In fact, given the syntactic restriction on the acceptable types of clausal interpretations, the explicit lowering operator $\lightning$ is unnecessary. If desired, closure, too, may be type-driven, at least for any operator $\lightning :: \Theta\upsilon \to \xi$ where ξ is ordinary (effect-free). The reason is that such operators reduce the complexity of their prejacents. They only apply to denotations with particular computational signatures, and in so doing, they strip off the types that trigger them. This guarantees that $\lightning$ can only apply to its own output finitely many times. In other words, no loops. Incorporating this into the type-driven framework of Figure 10, we might add meta-combinators like (5.18) for any appropriately typed closure operators $\lightning$.

(5.18) $\mathbf{D} :: (\sigma \to \tau \to \Theta\upsilon) \to \sigma \to \tau \to$
$\mathbf{D}\,(*)\,E_1\,E_2 \coloneqq \lightning\,(E_1 * E_2)$

Suppose that two constituents $E_1 :: \sigma$ and $E_2 :: \tau$ can be combined via $(*)$ to make something of type $\Theta\upsilon$. And further suppose that $\Theta\upsilon$ is a type that can be closed — a complete thought, so to speak — by an operator $\lightning :: \Theta\upsilon \to \xi$. For instance, if $\Theta = \mathtt{C}$ and $\upsilon = \mathtt{t}$, then the computation is ripe for lowering via

(5.15), producing an ordinary proposition of type `t`. In that case, the rule in (5.18) combines and lowers the two constituents E_1 and E_2 in one fell swoop, and the de-continuized composition continues.

5.3 Implementing adjoint effects in the type-driven interpreter

All the scaffolding for adding adjoint and closure operators has already been laid. As ever, we first add variants to the `Mode` representing the **E** and **D** operations.

```
data Mode
  = FA | BA | PM       -- etc
  | MR Mode | ML Mode  -- map right and map left
  | AP Mode            -- structured app
  | UR Mode | UL Mode  -- unit right and unit left
  | JN Mode            -- join
  | EP Mode            -- co-unit
  | DN Mode            -- closure
```

Then we add a predicate `adjoint` characterizing the relation between adjoint effects. The only adjunction we treat here is that between `W` and `G`, which are adjoint so long as they read and write the same kind of data. Thus we check that the parameters to the `W` and `G` effects are the same.

```
functor, applicative, monad :: EffX -> Bool
functor _          = True
applicative (WX s) = monoid s
applicative f      = functor f && True
monad f            = applicative f && True

monoid :: Ty -> Bool
monoid T = True
monoid _ = False

adjoint :: EffX -> EffX -> Bool
adjoint (WX i) (GX j) = i == j
adjoint _ _           = False
```

The `combine` rule is extended with two cases, one binary and one unary. The binary rule `addEP` looks for a pair of daughters `Comp f s` and `Comp g t` that have adjoint effects. When it finds them, it attempts to combine their underlying

types `s` and `t`. For every way `(op, u)` that these types can be combined, a top-level combinator `Ep op` is returned with result type `u`.

The unary rule `addDN` looks at the results of a binary combination to see whether it can be closed. For illustrative purposes, we include only the closure operator for continuation effects, that of Section 5.2. Actually we slightly generalize this to allow for lowering a quantifier at any location where it would be well-typed to do so. That is, we allow for lowering whenever we have reached a quantificational type $(\alpha \to \alpha) \to o$ that might be applied to the identity function, and thus closed. Checking that this is possible is just a matter of checking that the underlying type and intermediate parameter of the `C` effect are the same.

```
combine :: Ty -> Ty -> [(Mode, Ty)]
combine l r = binaryCombs >>= unaryCombs
  where
    binaryCombs =
      modes l r
      ++ addMR l r
      ++ addML l r
      ++ addAP l r
      ++ addUR l r
      ++ addUL l r
      -- if the left and right daughters are adjoint, try
      -- to cancel them out with their co-unit
      ++ addEP l r
    unaryCombs e =
      return e
      ++ addJN e
      -- and if the result type is close-able, close it
      ++ addDN e

addEP l r = case (l, r) of
  (Comp f s, Comp g t) | adjoint f g
    -> [ (EP op, u) | (op, u) <- combine s t ]
  _ -> [                                     ]

addDN e = case e of
  (op, Comp (CX o a') a) | a == a'
    -> [ (DN op, o) ]
  _ -> [            ]
```

To complete the picture on closure, we implement the discussion of island

enforcement from Section 5.2. Again the goal is just to give a proof of concept, showing how effect types can be used to do syntactic work. So we assume that islands correspond to particular syntactic nodes, and that the parser will one way or another have done the work of identifying such boundaries before the type-driven interpreter sets to work. We thus extend the `Syn` type to include a branching node that has been identified as an island for quantifier scopes.

```
data Syn
  = Leaf Ty String
  | Branch Syn Syn
  | Island Syn Syn
```

Next we have to specify which types the island seeks to trap. We do this with the predicate `evaluated`. `C`-type computations are obviously out. These are denotations with quantifiers that are still up in the air, consuming their contexts, so they do not count as `evaluated`. Other effects are passed over, but we recurse into their underlying types to make sure that they are not hiding any embedded un-lowered quantifiers. Likewise with function types, we check that they are not going to spring into life as newly escaped quantifiers as soon as they are saturated with an argument down the road. That leaves only atomic types, which all count as `evaluated`.

```
evaluated :: Ty -> Bool
evaluated t = case t of
  Comp (CX _ _) _ -> False
  Comp _ a        -> evaluated a
  _ :-> a         -> evaluated a
  _               -> True
```

We put this predicate to work when extending the interpreter `synsem`. The first two cases are the same as before. All that remains is to say how `Island` nodes are interpreted. And here, all we do is interpret them as if they were ordinary branching nodes, and then discard any unevaluated results. That is, any `e` among the results of `synsem (Branch lsyn rsyn)` must have a properly `evaluated` type to escape the island. And that's it!

```
synsem :: Syn -> [Sem]
synsem syn = case syn of
  (Leaf t w)          -> [Lex t w]
  (Branch lsyn rsyn) ->
    [ Comb ty op lsem rsem
      | lsem     <- synsem lsyn
      , rsem     <- synsem rsyn
      , (op, ty) <- combine (getType lsem) (getType rsem) ]
  (Island lsyn rsyn) ->
    [ e | e <- synsem (Branch lsyn rsyn), evaluated (getType e) ]
  where
    getType (Comb ty _ _ _) = ty
```

Appendix A

Implementations of combinatoric operations

A1 Types

Here we give Haskell encodings of the effect types used in this Element. As indicated in the main text, many of these are reproductions of or equivalent to data types that Haskell pre-defines. We note such correspondences in the comments.

```
data G r a = G (r -> a)          -- Haskell's `Reader`
data W w a = W (a, w)            -- Haskell's `Writer`
data M a   = Just a | Nothing    -- Haskell's `Maybe`
data T g a = T (g -> (a, g))     -- Haskell's `State`
data D g a = D (g -> [(a, g)])   -- Haskell's `StateT []`
data C o a = C ((a -> o) -> o)   -- Haskell's `Cont`
type S     = [ ]                 -- Haskell's `[]`
data F a   = F (a, [a])
```

A2 Functor instances

For each effect, we define a law-abiding `fmap` implementing the (•) operation assumed in Chapter 2.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor (G r) where
  fmap k (G m) =
    G (\g -> k (m g))

instance Functor (W w) where
  fmap k (W m) =
    W (k (fst m), snd m)

instance Functor M where
  fmap k m =
    case m of
      Just a -> Just (k a)
```

```
      Nothing -> Nothing

instance Functor (T i) where
  fmap k (T m) =
    T (\g -> let (a, h) = m g in (k a, h))

instance Functor (D s) where
  fmap k (D m) =
    D (\g -> let outs = m g in [(k a, h) | (a, h) <- outs])

instance Functor (C o) where
  fmap k (C m) =
    C (\c -> m (\a -> c (k a)))

instance Functor S where
  fmap k m
    = [k a | a <- m]

instance Functor F where
  fmap k (F m) =
    F (k (fst m), [k a | a <- snd m])
```

A3 Applicative instances

For each effect, we define a law-abiding `<*>`, implementing the $(\circledast)$ operation of Chapter 3.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

instance Applicative (G r) where
  pure x = G (\g -> x)
  (G ff) <*> (G xx) =
    G (\g -> ff g (xx g))

instance Monoid w => Applicative (W w) where
  pure x = W (x, mempty)
  (W ff) <*> (W xx) =
    W (fst ff (fst xx), snd ff <> snd xx)
```

```
instance Applicative M where
  pure x = Just x
  ff <*> xx =
    case (ff, xx) of
      (Just f, Just x) -> Just (f x)
      (_     , _     ) -> Nothing

instance Applicative (T g) where
  pure x = T (\g -> (x, g))
  (T ff) <*> (T xx) =
    T (\g -> let (f, h) = ff g
                 (x, i) = xx h
             in (f x, i))

instance Applicative (D g) where
  pure x = D (\g -> [(x, g)])
  (D ff) <*> (D xx) =
    D (\g -> [(f x, i) | (f, h) <- ff g, (x, i) <- xx h])

instance Applicative (C o) where
  pure x = C (\c -> c x)
  (C ff) <*> (C xx) =
    C (\c -> ff (\f -> xx (\x -> c (f x))))

instance Applicative S where
  pure x = [x]
  ff <*> xx
    = [f x | f <- ff, x <- xx]

instance Applicative F where
  pure x = F (x, [x])
  (F ff) <*> (F xx) =
    F (fst ff (fst xx), [f x | f <- snd ff, x <- snd xx])
```

A4 Monad instances

For each effect, we define a law-abiding `>>=`, implementing the (≫=) operation of Chapter 4.

```
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b
```

```
  return :: a -> f a
  return = pure

  join :: f (f a) -> f a
  join m = m >>= id

instance Monad (G g) where
  (G m) >>= k =
    G (\g -> let G n = k (m g) in n g)

instance Monoid w => Monad (W w) where
  (W m) >>= k =
    W (let W n = k (fst m) in (fst n, snd m <> snd n))

instance Monad M where
  m >>= k =
    case m of
      Just a  -> k a
      Nothing -> Nothing

instance Monad (T g) where
  (T m) >>= k =
    T (\g -> let (a, h) = m g
                 T n = k a
             in n h)

instance Monad (D g) where
  (D m) >>= k =
    D (\g -> [(b, i) | (a, h) <- m g
                     , let D n = k a
                     , (b, i) <- n h])

instance Monad (C o) where
  (C m) >>= k =
    C (\c -> m (\x -> let C n = k x in n c))

instance Monad S where
  m >>= k
    = [b | x <- m, b <- k x]

instance Monad F where
```

```
  (F m) >>= k =
    F ( let F n = k (fst m) in fst n
      , [b | a <- snd m, let F n = k a, b <- snd n])
```

A5 Adjunction instances

Here we implement the adjunction between `W` and `G`, relied upon in Chapter 5.

```
class Adjoint f g where
  unit :: a -> f (g a)
  counit :: g (f a) -> a

instance Adjoint (G g) (W g) where
  unit a = G (\g -> W (a, g))
  counit (W (G m, g)) = m g
```

Appendix B

The complete type-driven interpreter

Here we collect the snippets defined in the text implementing type-driven interpretation. The grammar is that of Figure 10, plus the **D** rule of (5.18).

```
-- representations of types
data Ty
  = TyE | TyT      -- primitive types
  | Ty :-> Ty      -- function types
  -- other compound types, as desired
  | Comp EffX Ty   -- computation types
  deriving (Eq, Show)

-- representations of effect constructors
data EffX
  = SX        -- computations with indeterminate results
  | GX Ty     -- computations that query an environment of type Ty
  | WX Ty     -- computations that store information of type Ty
  | CX Ty Ty  -- computations that quantify over Ty contexts
  -- and so on for other effects, as desired
  deriving (Eq, Show)

-- predicates characterizing the algebraic properties of the EffX
functor, applicative, monad :: EffX -> Bool
functor _          = True
applicative (WX s) = monoid s
applicative f      = functor f && True
monad f            = applicative f && True

monoid :: Ty -> Bool
monoid TyT = True
monoid _   = False

adjoint :: EffX -> EffX -> Bool
adjoint (WX i) (GX j) = i == j
adjoint _ _           = False

-- an inventory of combinatory modes
data Mode
```

```
  = FA | BA | PM       -- basic modes
  | MR Mode | ML Mode  -- map right and map left
  | AP Mode            -- structured app
  | UR Mode | UL Mode  -- unit right and unit left
  | JN Mode            -- join
  | EP Mode            -- co-unit
  | DN Mode            -- closure

-- syntactic objects to be interpreted
data Syn
  = Leaf Ty String
  | Branch Syn Syn
  | Island Syn Syn

-- semantic objects describing an interpretation
data Sem
  = Lex Ty String
  | Comb Ty Mode Sem Sem

-- the recursive interpreter
synsem :: Syn -> [Sem]
synsem syn = case syn of
  (Leaf t w)         -> [Lex t w]
  (Branch lsyn rsyn) ->
    [ Comb ty op lsem rsem
      | lsem     <- synsem lsyn
      , rsem     <- synsem rsyn
      , (op, ty) <- combine (getType lsem) (getType rsem) ]
  (Island lsyn rsyn) ->
    [ e | e <- synsem (Branch lsyn rsyn), evaluated (getType e) ]
  where
    getType (Comb ty _ _ _) = ty
    evaluated t = case t of
      Comp (CX _ _) _ -> False
      Comp _ a        -> evaluated a
      _ :-> a         -> evaluated a
      _               -> True

-- basic modes of combination
modes :: Ty -> Ty -> [(Mode, Ty)]
modes l r = case (l, r) of
  (a :-> b    , _      ) | r == a -> [(FA, b)]
```

```
  (_          , a :-> b) | l == a -> [(BA, b)]
  (TyE :-> TyT, TyE :-> TyT)      -> [(PM, TyE :-> TyT)]
  _                               -> []

-- the logic of applying higher-order modes to types
combine :: Ty -> Ty -> [(Mode, Ty)]
combine l r = binaryCombs >>= unaryCombs
  where
    binaryCombs =
      modes l r
      ++ addMR l r
      ++ addML l r
      ++ addAP l r
      ++ addUR l r
      ++ addUL l r
      ++ addEP l r
    unaryCombs e =
      return e
      ++ addJN e
      ++ addDN e

addMR, addML, addAP, addUR, addUL, addEP :: Ty -> Ty -> [(Mode, Ty)]
-- if the right daughter is functorial, try to map over it
addMR l r = case r of
  Comp f t | functor f
    -> [ (MR op, Comp f u) | (op, u) <- combine l t ]
  _ -> [                                            ]

-- if the left daughter is functorial, try to map over it
addML l r = case l of
  Comp f s | functor f
    -> [ (ML op, Comp f u) | (op, u) <- combine s r ]
  _ -> [                                            ]

-- if both daughters are applicative, try structured application
addAP l r = case (l, r) of
  (Comp f s, Comp g t) | f == g, applicative f
    -> [ (AP op, Comp f u) | (op, u) <- combine s t ]
  _ -> [                                            ]

-- if the left daughter closes an applicative effect,
-- try to purify the right daughter
```

```
addUR l r = case l of
  Comp f s :-> s' | applicative f
    -> [ (UR op, u) | (op, u) <- combine (s :-> s') r ]
  _ -> [                                              ]

-- if the right daughter closes an applicative effect,
-- try to purify the left daughter
addUL l r = case r of
  Comp f t :-> t' | applicative f
    -> [ (UL op, u) | (op, u) <- combine l (t :-> t') ]
  _ -> [                                              ]

-- if the left and right daughters are adjoint, try co-unit
addEP l r = case (l, r) of
  (Comp f s, Comp g t) | adjoint f g
    -> [ (EP op, u) | (op, u) <- combine s t ]
  _ -> [                                     ]

addJN, addDN :: (Mode, Ty) -> [(Mode, Ty)]
-- if a result of combination has a two-layered, join it
addJN e = case e of
  (op, Comp f (Comp g a)) | f == g, monad f
    -> [ (JN op, Comp f a) ]
  _ -> [                   ]

-- if a result of combination can be closed, close it
addDN e = case e of
  (op, Comp (CX o r) a) | r == a
    -> [ (DN op, o) ]
  _ -> [            ]
```

References

Ades, A. E., & Steedman, M. J. (1982). On the order of words. *Linguistics and philosophy*, *4*(4), 517–558.

Alonso-Ovalle, L. (2009). Counterfactuals, correlatives, and disjunction. *Linguistics and Philosophy*, *32*(2), 207–244.

Asudeh, A., & Giorgolo, G. (2020). *Enriched meanings: Natural language semantics with category theory* (Vol. 13). Oxford University Press.

Barker, C. (2002). Continuations and the nature of quantification. *Natural Language Semantics*, *10*(3), 211–242.

Barker, C., & Shan, C.-c. (2014). *Continuations and natural language* (Vol. 53). Oxford University Press.

Beaver, D., & Krahmer, E. (2001). A partial account of presupposition projection. *Journal of Logic, Language and Information*, *10*, 147–82.

Charlow, S. (2014). *On the semantics of exceptional scope* (PhD Dissertation). New York University, New York, NY.

Charlow, S. (2018). A modular theory of pronouns and binding. In *Logic and engineering of natural language semantics LENLS 14.*

Charlow, S. (2020). The scope of alternatives. *Linguistics and Philosophy*, *43*, 427–472.

Charlow, S. (2022). On jacobson's "towards a variable-free semantics". In L. McNally & Z. G. Szabó (Eds.), *A reader's guide to classic papers in formal semantics* (Vol. 100, pp. 171–196). Springer International Publishing. doi: 10.1007/978-3-030-85308-2_10

Chung, S., & Ladusaw, W. A. (2003). *Restriction and saturation*. MIT press.

Church, A. (1940). A formulation of the simple theory of types. *The journal of symbolic logic*, *5*(2), 56–68.

de Groote, P. (2001). Type raising, continuations, and classical logic. In R. van Rooij & M. Stokhof (Eds.), *13th Amsterdam Colloquium* (p. 97-101).

Dowty, D. (1988). Type raising, functional composition, and non-constituent conjunction. In *Categorial grammars and natural language structures* (pp. 153–197). Springer.

Eijck, J. v. (2001). Incremental dynamics. *Journal of Logic, Language and Information*, *10*(3), 319–351.

Goldstein, S. (2019). Free choice and homogeneity. *Semantics and Pragmatics*, *12*, 23.

Groenendijk, J., & Stokhof, M. (1984). *Studies on the semantics of questions and the pragmatics of answers* (Unpublished doctoral dissertation). Universiteit van Amsterdam.

Groenendijk, J., & Stokhof, M. (1991). Dynamic predicate logic. *Linguistics and Philosophy*, *14*, 39–100.

Hagstrom, P. (1998). *Decomposing questions* (PhD Dissertation). Massachusetts Institute of Technology, Cambridge, MA.

Hamblin, C. L. (1973). Questions in Montague English. *Foundations of Language*, *10*(1), 41-53.

Heim, I. (1982). *The semantics of definite and indefinite noun phrases* (Ph.D. Dissertation). University of Massachusetts, Amherst.

Heim, I., & Kratzer, A. (1998). *Semantics in Generative Grammar*. Oxford: Blackwell.

Jacobson, P. (1999). Towards a variable-free semantics. *Linguistics and Philosophy*, *22*(2), 117-184. doi: 10.1023/A:1005464228727

Jacobson, P. I. (2014). *Compositional semantics: An introduction to the syntax/semantics interface*. Oxford Textbooks in Linguistic.

Kamp, H. (1975). Two theories about adjectives. In *Formal semantics of natural language* (pp. 123–155).

Kiselyov, O. (2015). Applicative abstract categorial grammars in full swing. In *Jsai international symposium on artificial intelligence* (pp. 66–78).

Kiselyov, O., & Shan, C.-c. (2014). Continuation hierarchy and quantifier scope. In *Formal approaches to semantics and pragmatics* (pp. 105–134). Springer.

Klein, E., & Sag, I. A. (1985). Type-driven translation. *Linguistics and Philosophy*, 163–201.

Kratzer, A. (1996). Severing the external argument from its verb. In *Phrase structure and the lexicon* (pp. 109–137). Springer.

Kratzer, A., & Shimoyama, J. (2002). Indeterminate pronouns. In Y. Otsu (Ed.), *Third Tokyo conference on psycholinguistics* (p. 1-25). Tokyo. doi: 10.1007/978-3-319-10106-4_7

Krifka, M. (1995). The semantics and pragmatics of polarity items. *Linguistic Analysis*, *25*(3-4), 209–257.

Lewis, D. (1975). Adverbs of quantification. In E. Keenan (Ed.), *Formal semantics of natural language* (p. 3-15). Cambridge, MA: Cambridge University Press.

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22nd acm sigplan-sigact symposium on principles of programming languages* (pp. 333–343).

McBride, C., & Paterson, R. (2008). Applicative programming with effects. *Journal of functional programming*, *18*(1), 1–13.

Muskens, R. (1990). Anaphora and the logic of change. In *European workshop on logics in artificial intelligence* (pp. 412–427).

Muskens, R. (1996). Combining Montague semantics and Discourse Representation. *Linguistics and Philosophy*, *19*(2), 143–186.

Partee, B. (1986). Noun phrase interpretation and type-shifting principles. In J. Groenendijk, D. de Jongh, & M. Stokhof (Eds.), *Studies in Discourse Representation Theory and the theory of generalized quantifiers* (p. 115-144). Dordrecht: Foris. doi: 10.1002/9780470751305.ch10

Poesio, M. (1996). Semantic ambiguity and perceived ambiguity. In K. van Deemter & S. Peters (Eds.), *Semantic ambiguity and underspecification* (Vol. 55, pp. 159–201). Stanford: CSLI Publications.

Romero, M., & Novel, M. (2013). Variable binding and sets of alternatives. In *Alternatives in semantics* (pp. 174–208). Springer.

Rooth, M. (1985). *Association with focus* (PhD Dissertation). University of Massachusetts, Amherst, Amherst, MA.

Shan, C.-c. (2001a). Monads for natural language semantics. In K. Striegnitz (Ed.), *ESSLLI 2001 student session* (pp. 285–298).

Shan, C.-c. (2001b). A variable-free dynamic semantics. In R. van Rooy & M. Stokhof (Eds.), *Proceedings of the thirteenth Amsterdam Colloquium* (pp. 204–209). University of Amsterdam.

Shan, C.-c. (2005). *Linguistic side effects* (PhD Dissertation). Harvard University, Cambridge, MA.

Shimoyama, J. (2006). Indeterminate phrase quantification in Japanese. *Natural Language Semantics*, *14*, 139-173.

Siegel, M. E. A. (1976). *Capturing the adjective*. University of Massachusetts Amherst.

Vermeulen, C. F. M. (1993). Sequence semantics for dynamic predicate logic. *Journal of Logic, Language and Information*, *2*(3), 217–254.

Wadler, P. (1989). Theorems for free! In *Proceedings of the fourth international conference on functional programming languages and computer architecture* (pp. 347–359).

Acknowledgements