

# LINGUISTIC SIDE EFFECTS

CHUNG-CHIEH SHAN

HARVARD UNIVERSITY, 33 OXFORD STREET, CAMBRIDGE, MA 02138, USA

*E-mail address:* ccshan@post.harvard.edu

*URL:* <http://www.digitas.harvard.edu/~ken>

ABSTRACT. Making a linguistic theory is like specifying a programming language: one typically devises a type system to delineate the acceptable utterances and a denotational semantics to explain their operational behavior. Via this connection, programming and natural language research can inform each other; in particular, computational side effects are intimately related to referential opacity in natural languages. To illustrate this link, I use continuations and composable contexts, concepts from the study of side effects in programming languages, to analyze quantification, a natural language phenomenon.

## 1. INTRODUCTION

1.1. **The goals of natural language semantics.** This paper is about computational linguistics, in the sense of applying insights from computer science to linguistics. Linguistics strives to scientifically explain empirical observations of natural language. Semantics, in particular, is concerned with phenomena such as the following. In (1) below, some sentences to the left *ENTAIL* their counterparts to the right, but others do not.

- (1)    a. Every student passed.     $\vdash$  Every diligent student passed.  
      b. No student passed.         $\vdash$  No diligent student passed.  
      c. A student passed.          $\not\vdash$  A diligent student passed.  
      d. Most students passed.     $\not\vdash$  Most diligent students passed.

The sentence in (2) is *AMBIGUOUS* between at least two readings. On one reading, the speaker must decline to run any spot that fails to substantiate *any* claims, whatever these claims may be. On another reading, there exist certain claims (anti-war ones, say) such that the speaker must decline to run any spot that fails to substantiate *these* particular claims.

- (2)    We must decline to run any spot that fails to substantiate certain claims.<sup>1</sup>

Finally, among the four sentences in (3), only (3a) is *ACCEPTABLE*. That is, only it can be used in idealized conversation. The unacceptability of the rest is notated with asterisks.

---

Thanks to Stuart Shieber, Chris Barker, Barbara Grosz, Aravind Joshi, Fernando Pereira, Avi Pfeffer, Norman Ramsey, Dylan Thurston, and the audiences at the Harvard AI Research Group, the Boston University Church Research Group, the 8th New England Programming Languages and Systems Symposium, the University of Vermont, and the University of Pennsylvania. This work is supported by the United States National Science Foundation under Grant IRI-9712068.

<sup>1</sup>This sentence is part of a statement made by the cable television company Comcast after it rejected an anti-war commercial hours before it was scheduled to air over its CNN channel on January 28, 2003.

- (3) a. No student liked any course.  
 b. \*Every student liked any course.  
 c. \*A student liked any course.  
 d. \*Most students liked any course.

Presumably, the linguistic entailments and non-entailments in (1) have to do with corresponding logical facts: if every student passed, then every diligent student passed. (It is important to distinguish between linguistic and logical entailment. The example sentences in (1) can just as well be in Mandarin Chinese instead of English: presumably, the sentence 每個學生都及格了 entails the sentence 每個用功的學生都及格了 because, if every student passed, then every diligent student passed.) Thus the typical linguistic theory specifies a semantics for natural language by TRANSLATING declarative sentences into logical statements with truth conditions. The linguistic entailment in (1a) holds, goes the theory, because the meanings—truth conditions—of the two sentences are such that any model that verifies the former also verifies the latter. Much work in natural language semantics aims in this way, as depicted in Figure 1 on the facing page, to explain the horizontal by positing the vertical. Often the target of the translation posited is some combination of the  $\lambda$ -calculus and predicate logic. This approach is reminiscent of programming language research where an ill-understood language (perhaps one with a complicating feature like exceptions) is studied by translation into a simpler language (without exceptions) that is better understood.

1.2. **Side effects in programming and natural languages.** Making a linguistic theory is like specifying a programming language: one typically devises a type system to delineate the acceptable utterances and a denotational semantics to explain their operational behavior. In this paper, I will argue by example that programming languages can inform linguistic theory. In particular, the treatment of REFERENTIAL OPACITY in programming language semantics can inform that in linguistics. I now explain the concept of referential opacity for both programming and natural languages, and why I think the analogy is useful.

Roughly speaking, referential opacity means that equals cannot be substituted for equals (Quine 1960). To take a famous example, even though the English phrases

- (4) a. the morning star  
 b. the evening star

both refer to Venus, the larger sentences

- (5) a. Bob thinks Alice saw the morning star.  
 b. Bob thinks Alice saw the evening star.

are not equivalent: because Bob may not know much astronomy, the sentences do not entail each other. Analogously, in the programming language C, even though the expressions

- (6) a. 2  
 b.  $x = 2$

both give the result 2, the larger expressions

- (7) a. 2,  $x + 1$   
 b.  $x = 2$ ,  $x + 1$

may well give different results, for example if the variable  $x$  starts off with the value 1.

In the vast majority of today's programming languages, situations like (7) abound, where equals cannot be substituted for equals. For some expressions, such as `print` and `goto` commands, it is not even obvious what values there are to speak of, not to mention

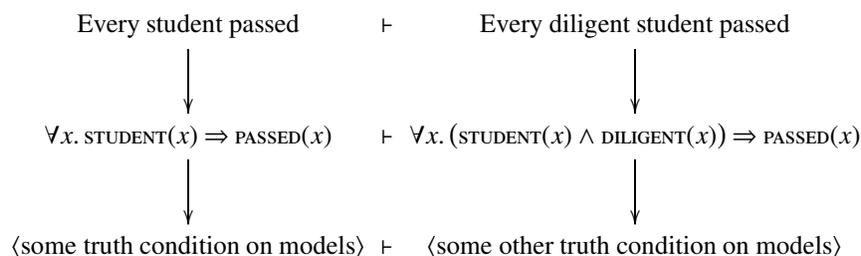


FIGURE 1. The translation/denotation approach to natural language semantics

compare for equality. These instances of referential opacity, called *computational side effects*, are exemplified in (8) below by expressions in a pidgin programming language. (Underlined are the language constructs that trigger computational side effects.)

- (8)
- |    |  |                  |
|----|--|------------------|
| a. | <u>read</u> + 10) <u>with</u> 2                          |                  |
|    | ‘Read the input and add it to 10, given the input 2.’    | (Input)          |
| b. | <u>print</u> 2; 10                                       |                  |
|    | ‘Print the number 2, then produce the number 10.’        | (Output)         |
| c. | x := 2; 10   |                  |
|    | ‘Store 2 in the variable x, then produce the number 10.’ | (State)          |
| d. | 2 + <u>random</u> (10, 20)                               |                  |
|    | ‘Add 2 to either 10 or 20, randomly chosen.’             | (Nondeterminism) |
| e. | <u>try</u> (2 + <u>throw</u> ) <u>catch</u> 3            |                  |
|    | ‘Add 2 to an error. Fall back to 3 in case of error.’    | (Exceptions)     |
| f. | <u>label</u> : 2 + <u>goto</u> label                     |                  |
|    | ‘Add 2 to the result of starting over again.’            | (Control)        |

In natural languages, referential opacity has been observed in a variety of settings, such as those in (9) below. By analogy to computer science, I term these *linguistic side effects*. They range from cases like (9a), where equals cannot be substituted for equals, to cases like (9d), where it is unclear what values there are to speak of and compare for equality. (Underlined are the constructions that trigger linguistic side effects.)

- (9)
- |    |   |                    |
|----|---|--------------------|
| a. | Bob <u>thinks</u> Alice saw the morning star.       | (Intensionality)   |
| b. | <u>A man</u> walks in the park. <u>He</u> whistles. | (Variable binding) |
| c. | <u>Every</u> woman whistles.                        | (Quantification)   |
| d. | <u>Which</u> star did Alice see?                    | (Interrogatives)   |
| e. | Alice <u>only</u> saw <u>VENUS</u> .                | (Focus)            |
| f. | <u>The king of France</u> whistles.                 | (Presuppositions)  |

To study linguistic side effects, I propose to draw an analogy between them and computational side effects, and investigate how linguists and computer scientists can take advantage of each other’s theoretical frameworks and empirical hypotheses. To start with, just as computer scientists want to express all computational side effects in a uniform framework and mix and match them at will, linguists want to investigate common properties shared by all linguistic side effects and study how they interact with each other. Furthermore, just as computer scientists want to relate operational notions such as EVALUATION ORDER and PARAMETER PASSING mechanism to denotational models such as MONADS and CONTINUATIONS, linguists want to relate the dynamics of information in language processing—for example

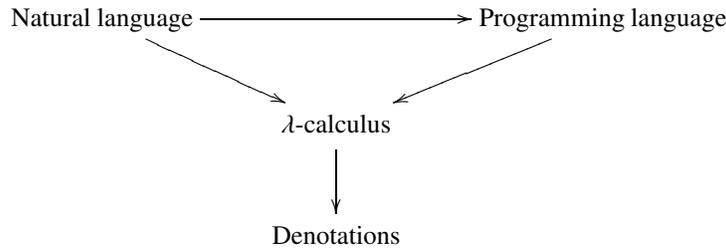


FIGURE 2. How programming language research can inform linguistic theory

how pronouns are generated by speakers and understood by hearers—to the static definition of a language as a generative device—for example what pronouns denote. If, instead of predicate logic in Figure 1 on the page before, we use a programming language with side effects as the middle tier of the translation, then referential opacity in natural language becomes easier to characterize and reason about. This claim of mine is depicted in Figure 2.

In this paper, I am mostly concerned with QUANTIFICATION, a particular linguistic side effect caused by words like *every* and *most* in (1). I show an analysis of quantification in terms of COMPOSABLE CONTEXTS that improves upon existing algorithms for quantifier scoping in terms of theoretical elegance and empirical coverage. However, as stated in the previous paragraph, a theory of linguistic side effects should uniformly handle all instances of referential opacity in natural languages as special cases of a general framework. Indeed, continuations and composable contexts can be used to characterize not just quantification (Barker 2002) but also variable binding and interrogatives (Shan and Barker 2003).

To complete the present introduction and make our understanding of Figure 2 more concrete, I take a brief look at variable binding in English. Empirical data such as those in (10) suggest that we think of pronouns like *she* as “load instructions” that retrieve referents previously “stored” by antecedents like *a student*.

- (10) a. A student passed. She was diligent.  $\vdash$  A student was diligent.  
 b. A professor praised a student. She was diligent. (Ambiguous)  
 c. She was diligent. A student passed. (Odd out of context)

Accordingly, we might posit translation rules that map (11a) to (11b).

- (11) a. Alice passed. She was diligent.  
 b. `PASSED(store(ALICE))  $\wedge$  DILIGENT(load())`

Here `store` and `load` are “side effect” constructs in the target language whose meanings then need to be clarified. It is not essential that this “storage” feature be present in the target language, since it can be regarded as mere syntactic sugar for the  $\lambda$ -calculus, just as the  $\lambda$ -calculus in turn is regarded by many linguists as syntactic sugar for set-theoretic denotations. Nevertheless, computational intuition can make for a linguistic theory that is easier to construct, understand, and maintain. Whether it also makes for a theory that is empirically adequate is a scientific question that I find attractive to pursue.

The rest of this paper is organized as follows. In §2, I specify a simple grammatical formalism. In §3, I describe the linguistic phenomenon of quantification and show a straw man analysis that deals with some cases but not others. I then introduce a programming

language with composable contexts and use it to improve the straw man analysis. In particular, quantification in non-subject position will be treated in §4, and inverse scope will be treated in §5. In §6, I place this example in a broader context and conclude.

## 2. A SIMPLE GRAMMATICAL FORMALISM

In this section, I specify a simple grammatical formalism for use in the rest of the paper. It is a notational variant of categorial grammar (as introduced in, for instance, Carpenter 1997, chapter 4).

The verb *like* usually requires an object to its right and a subject to its left.

- (12) a. Alice liked CS187.  
 b. \*Alice liked.  
 c. \*Alice liked Bob CS187.

Intuitively, the verb *like* takes two arguments, and the sentences (12b–c) are unacceptable due to type mismatch. We can model this formally by assigning types to the denotations of *Alice*, *CS187*, and *liked*, which we take to be atomic expressions.

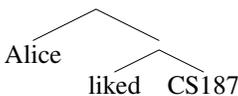
- (13)  $\llbracket \text{Alice} \rrbracket = \text{ALICE} : \text{THING}$   
 (14)  $\llbracket \text{CS187} \rrbracket = \text{CS187} : \text{THING}$   
 (15)  $\llbracket \text{liked} \rrbracket = \text{LIKED} : \text{THING} \rightarrow \text{THING} \rightarrow \text{BOOL}$

Here *THING* is the type of individual objects, and *BOOL* is the type of truth values or propositions. Following standard practice in linguistics, we let *LIKED* take its object as the first *THING* argument and its subject as the second *THING* argument. For example, in (12a), the first argument to *LIKED* is *CS187*, and the second argument is *ALICE*.

As (12a) shows, there are two ways in which smaller expressions are combined to form larger ones. A function can take its argument either to its right (combining *liked* with *CS187*) or to its left (combining *Alice* with *liked CS187*). We denote these two cases with two infix operators: “*⋅*” for forward combination and “*⋈*” for backward combination. (The tick marks can be thought of as depicting the direction in which a function “leans on top of” an argument.)

- (16)  $f \cdot x = f(x) : \beta$  where  $f : \alpha \rightarrow \beta$ ,  $x : \alpha$   
 (17)  $x \cdot f = f(x) : \beta$  where  $f : \alpha \rightarrow \beta$ ,  $x : \alpha$

We can now derive the sentence (12a)—that is, prove it to have type *BOOL*. The derivation can be written as a tree (18) or a term (19).

- (18) 

- (19)  $\llbracket \text{Alice} \rrbracket \cdot (\llbracket \text{liked} \rrbracket \cdot \llbracket \text{CS187} \rrbracket) = \text{LIKED}(\text{CS187})(\text{ALICE}) : \text{BOOL}$

By convention, the infix operators *⋅* and *⋈* associate to the right, so parentheses such as those in  $\llbracket \text{Alice} \rrbracket \cdot (\llbracket \text{liked} \rrbracket \cdot \llbracket \text{CS187} \rrbracket)$  above could have been elided and will be below.

Unfortunately, the system set up so far derives not only the acceptable sentence (12a) but also the unacceptable sentence (20), with the same meaning.

- (20) \*Alice CS187 liked.

The reason the system derives (20) is that the direction of function application is unconstrained: in the derivation below, *liked* takes its first (object) argument to the left, which is usually disallowed in English.

(21)

(22) 
$$\llbracket \text{Alice} \rrbracket \setminus \llbracket \text{CS187} \rrbracket \setminus \llbracket \text{liked} \rrbracket = \text{LIKED}(\text{CS187})(\text{ALICE}) : \text{BOOL}$$

To rule out this derivation of (20) in our type system, we split the function type constructor “ $\rightarrow$ ” into two type constructors “ $\dot{\rightarrow}$ ” and “ $\dot{\leftarrow}$ ”, one for each direction of application. Using these new type constructors, we change the denotation of *liked* to specify that its first argument is to its right and its second argument is to its left.

(23) 
$$\llbracket \text{liked} \rrbracket = \text{LIKED} : \text{THING} \dot{\leftarrow} \text{THING} \dot{\rightarrow} \text{BOOL}$$

We also revise the combination rules (16) and (17) to require different function type constructors.

(24) 
$$f \dot{\leftarrow} x = f(x) : \beta \quad \text{where } f : \alpha \dot{\leftarrow} \beta, x : \alpha$$
(25) 
$$x \setminus f = f(x) : \beta \quad \text{where } f : \alpha \dot{\rightarrow} \beta, x : \alpha$$

The system now rejects (20) while continuing to accept (12a), as desired.

### 3. QUANTIFICATION

The linguistic phenomenon of quantification is illustrated by the following sentences.

- (26) a. Every student liked CS187.  
 b. Some student liked every course.  
 c. Alice consulted Bob before most meetings.

As with the previously encountered English sentences, the natural language semanticist wants to translate these sentences into logical formulas that account for entailment and other properties. More precisely, the problem is to posit translation rules that map these sentences thus. For instance, we would like to map (26a) to a formula that looks like

(27) 
$$\forall x. \text{STUDENT}(x) \Rightarrow \text{LIKED}(\text{CS187})(x) : \text{BOOL}.$$

To this end, what should the subject *every student* denote? Unlike with *Alice*, there is nothing of type `THING` that the quantificational noun phrase *every student* can denote and still allow the desired translation (27) to be generated. At the same time, we would like to retain the denotation that we previously computed for the verb phrase *liked CS187*, namely  $\text{LIKED}(\text{CS187})$  in (19). Taking these considerations into account, one way to translate (26a) to (31) is for the determiner *every* to denote

(28) 
$$\llbracket \text{every} \rrbracket = \lambda r. \lambda s. \forall x. r(x) \Rightarrow s(x) : (\text{THING} \rightarrow \text{BOOL}) \dot{\leftarrow} (\text{THING} \dot{\rightarrow} \text{BOOL}) \dot{\rightarrow} \text{BOOL}.$$

Here the `RESTRICTOR`  $r$  and the `SCOPE`  $s$  are  $\lambda$ -bound variables intended to receive, respectively, the denotations of the noun *student* (of type `THING`  $\rightarrow$  `BOOL`) and the verb phrase *liked CS187* (of type `THING`  $\dot{\rightarrow}$  `BOOL`). In a non-quantificational sentence like (12a), the verb phrase takes the subject as its argument; by contrast, in the quantificational sentence (26a), the subject takes the verb phrase as its argument.

Extended with the lexical entry (28) for *every*, and assuming that *student* denotes

(29) 
$$\llbracket \text{student} \rrbracket = \text{STUDENT} : \text{THING} \rightarrow \text{BOOL},$$

the grammar now derives the sentence (26a).

(30)

$$\begin{array}{c}
 \diagup \quad \diagdown \\
 \text{every} \quad \text{student} \quad \text{liked} \quad \text{CS187} \\
 \diagdown \quad \diagup \quad \diagdown \quad \diagup \\
 \llbracket \text{every} \rrbracket \ / \ \llbracket \text{student} \rrbracket \ / \ \llbracket \text{liked} \rrbracket \ / \ \llbracket \text{CS187} \rrbracket = (27)
 \end{array}$$

(31)

The existential determiner *some* can be analyzed similarly. Specifically, let *some* denote

(32)  $\llbracket \text{some} \rrbracket = \lambda r. \lambda s. \exists x. r(x) \wedge s(x) : (\text{THING} \rightarrow \text{BOOL}) \rightarrow (\text{THING} \rightarrow \text{BOOL}) \rightarrow \text{BOOL}$

to derive *some student liked CS187*.

(33)

$$\begin{array}{c}
 \diagup \quad \diagdown \\
 \text{some} \quad \text{student} \quad \text{liked} \quad \text{CS187} \\
 \diagdown \quad \diagup \quad \diagdown \quad \diagup \\
 \llbracket \text{some} \rrbracket \ / \ \llbracket \text{student} \rrbracket \ / \ \llbracket \text{liked} \rrbracket \ / \ \llbracket \text{CS187} \rrbracket = \exists x. \text{STUDENT}(x) \wedge \text{LIKED}(\text{CS187})(x) : \text{BOOL}
 \end{array}$$

(34)

Let me summarize the basic idea behind this analysis. We treat determiners like *every* and *some* as functions of two arguments: the restrictor and the scope of a quantifier, both functions from `THING` to `BOOL`. Such higher-order functions are a popular analysis of natural language determiners, and have been known to semanticists since Montague (1974) as `GENERALIZED QUANTIFIERS`. However, the simplistic account presented above only handles quantificational noun phrases in subject position, as in (26a) but not (26b) or (26c). For example, in (26b), neither forward nor backward combination can apply to join the verb *liked*, of type `THING → THING → BOOL`, to its object *every course*, of type `(THING → BOOL) → BOOL`. Yet, empirically speaking, the sentence (26b) is not only acceptable but in fact ambiguous between two available readings. This problem has prompted a great variety of supplementary proposals in the linguistics literature (Barwise and Cooper 1981; Hendriks 1993; May 1985; inter alia). In the next section, I will present a particular solution using `COMPOSABLE CONTEXTS` that is closely related to Barker's (2002) continuations analysis.

#### 4. COMPOSABLE CONTEXTS

In this section, I introduce composable contexts in programming languages and use them to analyze quantification in natural languages. I begin with a simple programming language of numerical expressions.

(35)  $2 + 1 \implies 3$

(36)  $2 * 5 + 1 \implies 11$

Suppose we are interested in exceptions as a programming language feature. We may model exceptions in our toy language by introducing a new operator `abort`, which when evaluated causes the program to immediately terminate with a supplied value, as in

(37)  $2 * \text{abort}(5) + 1 \implies 5.$

To disambiguate expressions with more than one `abort`, we stipulate that evaluation proceeds from left to right. For instance,

(38)  $\text{abort}(5) + \text{abort}(8) \implies 5$  (rather than 8).

The `abort` operator allows exceptions to be thrown in a rudimentary fashion. To model the catching of exceptions, we introduce another operator `RESET`, notated by square brackets `[ ]`, which limits the extent to which `abort` can apply. For example,

$$(39) \quad [2 * \text{abort } (5)] + 1 \quad \Longrightarrow \quad 6 \quad (= 5 + 1),$$

because the effect of `abort` cannot reach beyond the reset. (The entire expression is always implicitly bracketed, in other words implicitly enclosed in a top-level reset.)

One way to study computational side effects such as exceptions, introduced by Danvy and Filinski (1990; Filinski 1996), is with reset and another operator called `shift`. The `shift` operator is similar to `abort` in that they both remove the current context of computation (up to the closest enclosing reset), but `shift` makes this context available to the program as a function. This behavior is best explained with examples. In the following expression, the variable `f` is bound to the function that multiplies every number by 10.

$$(40) \quad 10 * (\text{shift } f: 1 + f(2)) \quad \Longrightarrow \quad 21 \quad (= 1 + 10 \cdot 2).$$

To be more specific, the above expression evaluates to 21 via the following sequence of reductions. (The reduced subexpression at each step is underlined.)

$$(41) \quad \begin{aligned} & [10 * (\underline{\text{shift } f: 1 + f(2)})] \\ & \rightsquigarrow [1 + \underline{f(2)}] \quad \text{where } f \equiv \text{lambda } v: 10 * v \\ & \rightsquigarrow [1 + [10 * \underline{2}]] \\ & \rightsquigarrow [1 + [\underline{10 * 2}]] \\ & \rightsquigarrow [1 + [\underline{20}]] \rightsquigarrow [\underline{1 + 20}] \rightsquigarrow [\underline{21}] \rightsquigarrow 21 \end{aligned}$$

The reset brackets `[ ]` delimit not just how far `abort` can reach but also how far `shift` can reach. For example,

$$(42) \quad 10 * [3 + (\text{shift } f: f(0) + f(1))] \quad \Longrightarrow \quad 70 \quad (= 10 \cdot (3 + 0 + 3 + 1))$$

via the following reduction sequence.

$$(43) \quad \begin{aligned} & [10 * [3 + (\underline{\text{shift } f: f(0) + f(1)})]] \\ & \rightsquigarrow [10 * [\underline{f(0)} + f(1)]] \quad \text{where } f \equiv \text{lambda } v: 3 + v \\ & \rightsquigarrow [10 * [[3 + \underline{0}] + f(1)]] \\ & \rightsquigarrow [10 * [[\underline{3 + 0}] + f(1)]] \\ & \rightsquigarrow [10 * [[\underline{3}] + f(1)]] \\ & \rightsquigarrow [10 * [3 + \underline{f(1)}]] \\ & \rightsquigarrow [10 * [3 + [3 + \underline{1}]]] \rightsquigarrow [10 * [3 + [\underline{3 + 1}]]] \rightsquigarrow [10 * [3 + [\underline{4}]]] \\ & \rightsquigarrow [10 * [\underline{3 + 4}]] \rightsquigarrow [10 * [\underline{7}]] \rightsquigarrow [\underline{10 * 7}] \rightsquigarrow [\underline{70}] \rightsquigarrow 70 \end{aligned}$$

Danvy and Filinski not only specify an *operational* semantics for `shift` and reset (illustrated by the reduction sequences above), but also use continuations to spell out a *denotational* semantics for the two operators. The fact that both kinds of semantics are available and they are consistent with each other is important to linguistics, because the meanings of natural language expressions (according to the field of formal semantics) need to be related to how humans process them (according to the field of psycholinguistics). In this paper,

though, I will not be concerned with denotations.<sup>2</sup> Rather, I will focus on how composable contexts establish an analogy between computational and linguistic side effects.

On one hand, computational side effects such as exceptions, state, and input/output can be expressed in terms of `shift` and `reset` (Filinski 1996). For example, the `abort` operator can be treated as syntactic sugar for `shift f`, where the captured context `f` is never used.

$$(44) \quad \text{abort } (e) \equiv \text{shift } f: e.$$

The celebrated `call/cc` (call with current continuation) operator can also be expressed in terms of `shift`.

$$(45) \quad \text{call/cc } (e) \equiv \text{shift } f: f(e(\text{lambda } x: \text{shift } g: f(x))).$$

On the other hand, as Barker (2002) notes, quantificational phrases in natural language can be thought of as expressions that manipulate their context. In a sentence such as *Alice liked CS187* (12a), the context of *CS187* is the function mapping each thing  $x$  to the proposition that Alice liked  $x$ . Compared to the proper noun *CS187*, what is special about a quantificational expression like *every course* is that it captures its surrounding context when used.

$$(46) \quad \text{Alice liked [every course].}$$

Thus, loosely speaking, the meaning of the sentence (46) no longer has the overall shape `LIKED(...)(ALICE)` once the occurrence of *every course* is considered, much as the the meaning of the program (40) no longer has the overall shape `!0 * ...` once the `shift` expression is evaluated. As promised in Figure 2, then, let us add `shift` and `reset` to the target language of our translation from English. We can then translate *every course* as

$$(47) \quad \llbracket \text{every course} \rrbracket = \text{shift } s: \forall x. \text{COURSE}(x) \Rightarrow s(x) : \text{THING}.$$

This novel denotation for the natural language expression *every course*, inspired by `shift` and `reset` from programming language research, is closely related to Barker’s (2002) continuations analysis.<sup>3</sup>

To see the new denotation (47) in action, let us derive the sentence (46). Note that the type of *every course* is simply `THING`, the same as *CS187*, so the derivation of (46) is structurally analogous to (18–19).

$$(48) \quad \begin{array}{c} \diagup \quad \diagdown \\ \text{Alice} \quad \quad \quad \diagup \quad \diagdown \\ \quad \quad \quad \text{liked} \quad \text{every course} \end{array}$$

As with our toy programming language, we perform term reductions in applicative order (call-by-value and left-to-right). The crucial reduction is that of the `shift`-expression

<sup>2</sup>Briefly, for readers familiar with continuations, the denotation semantics is as follows.

$$\begin{aligned} \llbracket 2 \rrbracket &= \lambda c. c(2), \\ \llbracket e_1 + e_2 \rrbracket &= \lambda c. \llbracket e_1 \rrbracket (\lambda x. \llbracket e_2 \rrbracket (\lambda y. c(x + y))), \\ \llbracket \text{shift } f: e \rrbracket &= \lambda c. \llbracket e \rrbracket \{f \mapsto \lambda x c'. c'(cx)\}(\lambda x. x), \\ \llbracket [e] \rrbracket &= \lambda c. c(\llbracket e \rrbracket (\lambda x. x)). \end{aligned}$$

<sup>3</sup>The differences between the analysis presented here and Barker’s continuations analysis go beyond the use of `shift` and `reset` notation in this paper. For instance, Barker uses choice functions to deal with restrictors of quantifiers (such as *course* in *every course*), whereas I do not.

(from the fourth line to the fifth), as one might expect.

$$\begin{aligned}
(49) \quad & \llbracket \text{Alice} \rrbracket \setminus \llbracket \text{liked} \rrbracket \setminus \llbracket \text{every course} \rrbracket \\
& \equiv \llbracket (\lambda x f. f x)(\text{ALICE})((\lambda f x. f x)(\text{LIKED})(\text{shift } s : \forall x. \text{COURSE}(x) \Rightarrow s(x))) \rrbracket \\
& \rightsquigarrow \llbracket (\lambda f. f(\text{ALICE}))((\lambda f x. f x)(\text{LIKED})(\text{shift } s : \forall x. \text{COURSE}(x) \Rightarrow s(x))) \rrbracket \\
& \rightsquigarrow \llbracket (\lambda f. f(\text{ALICE}))((\lambda x. \text{LIKED}(x))(\text{shift } s : \forall x. \text{COURSE}(x) \Rightarrow s(x))) \rrbracket \\
& \rightsquigarrow \llbracket \forall x. \text{COURSE}(x) \Rightarrow \underline{s(x)} \rrbracket \quad \text{where } s \equiv \lambda v. (\lambda f. f(\text{ALICE}))((\lambda x. \text{LIKED}(x))v) \\
& \rightsquigarrow \llbracket \forall x. \text{COURSE}(x) \Rightarrow \llbracket (\lambda f. f(\text{ALICE}))((\lambda x. \text{LIKED}(x))(x)) \rrbracket \rrbracket \\
& \rightsquigarrow \llbracket \forall x. \text{COURSE}(x) \Rightarrow \llbracket (\lambda f. f(\text{ALICE}))(\text{LIKED}(x)) \rrbracket \rrbracket \\
& \rightsquigarrow \llbracket \forall x. \text{COURSE}(x) \Rightarrow \llbracket \text{LIKED}(x)(\text{ALICE}) \rrbracket \rrbracket \\
& \rightsquigarrow \llbracket \forall x. \text{COURSE}(x) \Rightarrow \text{LIKED}(x)(\text{ALICE}) \rrbracket \\
& \rightsquigarrow \forall x. \text{COURSE}(x) \Rightarrow \text{LIKED}(x)(\text{ALICE}) : \text{BOOL}
\end{aligned}$$

Like the straw man analysis in §3, the denotation in (47) generalizes to determiners other than *every*: we can abstract away from the noun *course* in *every course*, and deal with *some student* analogously.

$$(50) \quad \llbracket \text{every} \rrbracket = \lambda r. \text{shift } s : \forall x. [r(x)] \Rightarrow s(x) : (\text{THING} \rightarrow \text{BOOL}) \dot{\rightarrow} \text{THING}$$

$$(51) \quad \llbracket \text{some} \rrbracket = \lambda r. \text{shift } s : \exists x. [r(x)] \wedge s(x) : (\text{THING} \rightarrow \text{BOOL}) \dot{\rightarrow} \text{THING}$$

In these denotations, I wrapped the restrictor  $r(x)$  inside reset brackets. This technical detail is needed in case the restrictor itself invokes quantification, as in *every representative of some company*, where the restrictor is *representative of some company*.<sup>4</sup>

<sup>4</sup>Our grammar needs to be further extended before we can analyze a sentence like *Every representative of some company left*. First, we need lexical entries for two nouns and a verb.

$$\begin{aligned}
\llbracket \text{representative} \rrbracket &= \text{REPRESENTATIVE} : \text{THING} \rightarrow \text{BOOL}, \\
\llbracket \text{company} \rrbracket &= \text{COMPANY} \quad : \text{THING} \rightarrow \text{BOOL}, \\
\llbracket \text{left} \rrbracket &= \text{LEFT} \quad : \text{THING} \dot{\rightarrow} \text{BOOL}.
\end{aligned}$$

Second, and less trivially, we need the following denotation for the preposition *of*.

$$\llbracket \text{of} \rrbracket = \lambda x p y. p(y) \wedge \text{OF}(x(*))(y) : (* \rightarrow \text{THING}) \dot{\rightarrow} (\text{THING} \rightarrow \text{BOOL}) \dot{\rightarrow} \text{THING} \rightarrow \text{BOOL}.$$

The asterisk  $*$  here denotes both the unit type and its unique element. In the derivation below, we use the unit type to implement thunks (Hatchiff and Danvy 1997) in order to get around—that is, delay the side effects triggered by—call-by-value evaluation. We also add an important new way to build expressions from subexpressions: the denotation of any subexpression, such as *some company*, can be turned into a thunk by surrounding it with  $\lambda*$ .

$$\begin{aligned}
& \llbracket (\llbracket \text{every} \rrbracket \setminus \llbracket \text{representative} \rrbracket \setminus \llbracket \text{of} \rrbracket \setminus (\lambda*. \llbracket \text{some} \rrbracket \setminus \llbracket \text{company} \rrbracket)) \setminus \llbracket \text{left} \rrbracket \rrbracket \\
& \equiv \llbracket (\lambda x f. f x)((\lambda f x. f x)(\lambda r. \text{shift } s_1 : \forall x_1. [r(x_1)] \Rightarrow s_1(x_1)) \\
& \quad \llbracket (\lambda f x. f x)(\text{REPRESENTATIVE}) \rrbracket \\
& \quad \llbracket (\lambda f x. f x)(\lambda x p y. p(y) \wedge \text{OF}(x(*))(y)) \rrbracket \\
& \quad \llbracket (\lambda*. (\lambda f x. f x)(\lambda r. \text{shift } s_2 : \exists x_2. [r(x_2)] \wedge s_2(x_2))(\text{COMPANY})) \rrbracket \rrbracket \rrbracket (\text{LEFT}) \\
& \rightsquigarrow \dots \\
& \rightsquigarrow \llbracket (\lambda x f. f x)((\lambda x. (\lambda r. \text{shift } s_1 : \forall x_1. [r(x_1)] \Rightarrow s_1(x_1))x) \\
& \quad \llbracket (\lambda y. \text{REPRESENTATIVE}(y) \wedge \text{OF}(\lambda*. (\lambda f x. f x) \\
& \quad \quad \llbracket (\lambda r. \text{shift } s_2 : \exists x_2. [r(x_2)] \wedge s_2(x_2))(\text{COMPANY})(*) \rrbracket \rrbracket \rrbracket (y)) \rrbracket \rrbracket (\text{LEFT}) \\
& \rightsquigarrow \dots \\
& \rightsquigarrow \llbracket (\lambda x f. f x)(\text{shift } s_1 : \forall x_1. \llbracket (\lambda y. \text{REPRESENTATIVE}(y) \wedge \text{OF}(\lambda*. (\lambda f x. f x) \\
& \quad \quad \llbracket (\lambda r. \text{shift } s_2 : \exists x_2. [r(x_2)] \wedge s_2(x_2))(\text{COMPANY})(*) \rrbracket \rrbracket \rrbracket (y)) \rrbracket \rrbracket (x_1)) \rrbracket \rrbracket (\text{LEFT})
\end{aligned}$$

Moreover, unlike the straw man analysis, the present analysis works uniformly for quantificational expressions in subject, object, and other positions, such as in (26a–c). Intuitively, this is because the `shift` operator captures the context of an expression no matter how deeply it is embedded within the sentence. In sum, by drawing an analogy from computational side effects to linguistic ones, we have arrived at an analysis of quantification with greater empirical coverage.

## 5. HIGHER-ORDER CONTEXTS

Of course, natural language phenomena are never as simple as a couple of programming language operators. Quantification is no exception, so to speak. For example, consider a sentence like (26b) (repeated below), which contains multiple quantificational phrases.

(26b) Some student liked every course.

Like the sentence (2) from the introduction, this sentence is ambiguous between two readings. In the `SURFACE SCOPE` reading (52), *some* takes scope over *every*. In the `INVERSE SCOPE` reading (53), *every* takes scope over *some*.

$$(52) \quad \exists s. \text{STUDENT}(x) \wedge \forall c. \text{COURSE}(c) \Rightarrow \text{LIKED}(c)(s)$$

$$(53) \quad \forall c. \text{COURSE}(c) \Rightarrow \exists s. \text{STUDENT}(x) \wedge \text{LIKED}(c)(s)$$

Because we specified that evaluation takes place from left to right, our grammar predicts the surface scope reading but not the inverse scope reading. This prediction can be seen in the beginning of the (unique) derivation for (26b):

$$(54) \quad \begin{aligned} & [([\text{some}] \text{ / } [\text{student}]) \text{ / } [\text{liked}] \text{ / } [\text{every}] \text{ / } [\text{course}]] \\ & \equiv [(\lambda x f. f x)((\lambda f x. f x)(\lambda r. \text{shift } s: \exists x. [r(x)] \wedge s(x))(\text{STUDENT})) \\ & \quad ((\lambda f x. f x)(\text{LIKED}))((\lambda f x. f x)(\lambda r. \text{shift } s: \forall x. [r(x)] \Rightarrow s(x))(\text{COURSE})))] \\ & \rightsquigarrow [(\lambda x f. f x)(\underline{(\lambda x. (\lambda r. \text{shift } s: \exists x. [r(x)] \wedge s(x))x)(\text{STUDENT})}) \\ & \quad ((\lambda f x. f x)(\text{LIKED}))((\lambda f x. f x)(\lambda r. \text{shift } s: \forall x. [r(x)] \Rightarrow s(x))(\text{COURSE})))] \\ & \rightsquigarrow [(\lambda x f. f x)(\underline{(\lambda r. \text{shift } s: \exists x. [r(x)] \wedge s(x))(\text{STUDENT})}) \\ & \quad ((\lambda f x. f x)(\text{LIKED}))((\lambda f x. f x)(\lambda r. \text{shift } s: \forall x. [r(x)] \Rightarrow s(x))(\text{COURSE})))] \\ & \rightsquigarrow [(\lambda x f. f x)(\text{shift } s: \underline{\exists x. [\text{STUDENT}(x)] \wedge s(x)}) \\ & \quad ((\lambda f x. f x)(\text{LIKED}))((\lambda f x. f x)(\lambda r. \text{shift } s: \forall x. [r(x)] \Rightarrow s(x))(\text{COURSE})))] \\ & \rightsquigarrow [\exists x. [\text{STUDENT}(x)] \wedge s(x)] \quad \text{where } s \equiv \lambda v. (\lambda x f. f x)(v) \\ & \quad ((\lambda f x. f x)(\text{LIKED}))((\lambda f x. f x)(\lambda r. \text{shift } s: \forall x. [r(x)] \Rightarrow s(x))(\text{COURSE})))] \\ & \rightsquigarrow \dots \\ & \rightsquigarrow \exists x. \text{STUDENT}(x) \wedge \forall y. \text{COURSE}(y) \Rightarrow \text{LIKED}(y)(x). \end{aligned}$$

---


$$\begin{aligned} & \rightsquigarrow [\forall x_1. [\underline{(\lambda y. \text{REPRESENTATIVE}(y) \wedge \text{OF}(\lambda *. (\lambda f x. f x) \\ & \quad (\lambda r. \text{shift } s_2: \exists x_2. [r(x_2)] \wedge s_2(x_2))(\text{COMPANY})(*) (y)(x_1)) \Rightarrow s_1(x_1)}]} \\ & \quad \text{where } s_1 \equiv \lambda v. (\lambda x f. f x)(v)(\text{LEFT})] \\ & \rightsquigarrow \dots \\ & \rightsquigarrow [\forall x_1. [\text{REPRESENTATIVE}(x_1) \wedge \underline{\text{OF}(\text{shift } s_2: \exists x_2. [\text{COMPANY}(x_2)] \wedge s_2(x_2))(x_1)}] \Rightarrow s_1(x_1)] \\ & \rightsquigarrow [\forall x_1. [\exists x_2. [\text{COMPANY}(x_2)] \wedge s_2(x_2)] \Rightarrow s_1(x_1)] \quad \text{where } s_2 \equiv \lambda v. \text{REPRESENTATIVE}(x_1) \wedge \text{OF}(v)(x_1) \\ & \rightsquigarrow \dots \\ & \rightsquigarrow \forall x_1. (\exists x_2. \text{COMPANY}(x_2) \wedge \text{REPRESENTATIVE}(x_1) \wedge \text{OF}(x_2)(x_1)) \Rightarrow \text{LEFT}(x_1) : \text{BOOL} \end{aligned}$$

The `shift` for *some student* is evaluated before the `shift` for *every course*, so the former dictates the shape of the final result at the outermost level. Regardless of what evaluation order we specify, as long as our rules for semantic translation remain deterministic, they will only generate one reading for the sentence. In other words, our theory has failed to predict the ambiguity that is empirically observed in the sentence (26b).

To better account for the data, we need to introduce some sort of nondeterminism into our theory. There are two natural ways to proceed. First, we can allow arbitrary evaluation order, not just left-to-right. This change would render our term calculus nonconfluent, a result unwelcome for most programming language researchers but welcome for us in light of the ambiguous natural language sentence (26b). This route has been pursued with some success by Barker (2002) and de Groote (2001). However, for empirical reasons outside the scope of this paper (Shan and Barker 2003), I want to maintain the hypothesis that evaluation in natural language always proceeds from left to right. A second way to introduce nondeterminism into our semantic theory is to maintain that hypothesis but allow HIGHER-ORDER CONTEXTS (Barker 2000; Danvy and Filinski 1990).

In our toy programming language with `shift` and `reset`, the effect of `shift` is restricted to the closest enclosing `reset`. Higher-order contexts relax this restriction by placing a subscript on each `shift` and `reset` operator. The effect of an  $n$ th-order `shift` operator, written `shiftn`, is restricted to the closest enclosing  $m$ th-order `reset` operator, written `[ ]m`, such that  $m \geq n$ . Thus the subscripts can be thought of as indicating the “strength” of each `shift` and `reset`. For example,

$$(55) \quad \begin{array}{ll} 1 + [\text{shift}_1 f : 2]_1 \implies 3, & 1 + [\text{shift}_1 f : 2]_2 \implies 3, \\ 1 + [\text{shift}_2 f : 2]_1 \implies 2, & 1 + [\text{shift}_2 f : 2]_2 \implies 3. \end{array}$$

Just as they give a denotational semantics for first-order composable contexts using first-order continuations, Danvy and Filinski (1990) give a denotational semantics for higher-order composable contexts using higher-order continuations. We can take advantage of that work in our quantificational denotations (50–51) by allowing them to manipulate the context at any order, say the  $n$ th order.

$$(56) \quad \llbracket \text{every}_n \rrbracket = \lambda r. \text{shift}_n s : \forall x. [r(x)]_n \Rightarrow s(x) : (\text{THING} \rightarrow \text{BOOL}) \dot{\rightarrow} \text{THING}$$

$$(57) \quad \llbracket \text{some}_n \rrbracket = \lambda r. \text{shift}_n s : \exists x. [r(x)]_n \wedge s(x) : (\text{THING} \rightarrow \text{BOOL}) \dot{\rightarrow} \text{THING}$$

In other words, we posit that each occurrence of *every* and *some* be ambiguous as to the order  $n$  of its `shift` and `reset`.

The ambiguity of (26b) is now predicted as follows. Suppose that *some student* operates on the  $m$ th-order context and *every course* operates on the  $n$ th-order context.

$$(58) \quad \text{Some}_m \text{ student liked every}_n \text{ course.}$$

If  $m \geq n$ , the surface scope reading (52) results. If  $m < n$ , the inverse scope reading (53) results. The same treatment applies to more complicated cases of quantification in English and Mandarin Chinese (Shan 2003). For example, we predict correctly that the following sentence, with three quantifiers, is five-way (not six-way) ambiguous.

$$(59) \quad \text{Every representative of some company saw most samples.}$$

There exist in the computational linguistics literature algorithms for computing the possible quantifier scopings of a given sentence (Hobbs and Shieber 1987; followed by Lewin 1990; Moran 1988). Higher-order composable contexts provide a denotational understanding of these algorithms that accords with our theoretical intuitions and empirical observations.

Functor	Type-lifting map	Computational s.e.	Linguistic s.e.
Exponential	$\tau \rightarrow -$	•	• <sup>1</sup> • <sup>2</sup>
Product	$\tau \times -$	•	• <sup>3</sup>
State	$\tau_1 \rightarrow (\tau_2 \times -)$	•	• <sup>4</sup>
Powerset	$\{X \mid X \subseteq -\}$	•	• <sup>3</sup> • <sup>5</sup>
Pointed powerset	$\{\langle x, X \rangle \mid x \in X \subseteq -\}$		• <sup>6</sup>
Sum	$\tau + -$	•	• <sup>7</sup>
Continuation	$(- \rightarrow \tau_1) \rightarrow \tau_2$	• • • • •	• <sup>8</sup> • <sup>9</sup> • <sup>10</sup> • <sup>4</sup>

Input

Output

State

Non-determinism

Exceptions

Control

Intensionality

Variable binding

Quantification

Interrogatives

Focus

Presuppositions

<sup>1</sup>Montague 1974 <sup>2</sup>Jacobson 1999; Montague 1974 <sup>3</sup>Shan 2001b <sup>4</sup>Shan 2001a <sup>5</sup>Hamblin 1973 <sup>6</sup>Rooth 1985 <sup>7</sup>Spivey 1990 <sup>8</sup>Shan and Barker 2003 <sup>9</sup>Barker 2001; de Groot 2001; Hendriks 1993; Montague 1974 <sup>10</sup>Shan 2002

TABLE 1. Functors implicated in computational and linguistic side effects (see also (8) and (9) on page 3)

## 6. BEYOND QUANTIFICATION

As I stated in §1.2, modeling quantification in natural language with composable contexts is part of a larger project, that of relating computational side effects to linguistic ones. Three themes are crucial to this project: UNIFORMITY, INTERACTION, and EVALUATION. I briefly consider each in turn.

**6.1. Uniformity.** Most of the linguistics literature treats each side effect as an individual topic of study, for example as a separate chapter of a semantics textbook or as a separate rule or set of rules for semantic combination. By contrast, I aim to cover all linguistic side effects as special cases of a single, general framework. Computer science has developed several uniform models of side effects, including MONADS (Moggi 1991; Wadler 1995; inter alia) and CONTINUATIONS (Danvy and Filinski 1990; Filinski 1996; inter alia). Monads and continuations are both FUNCTORS, a kind of type-lifting operations on denotations that is also the standard strategy in linguistics for dealing with side effects. Table 1 lists the most commonly used functors and shows potential common uses across natural and programming languages. This table makes clear that many functors of interest are shared—a promising sign.

To illustrate that computer science can help us create a uniform theory of linguistic side effects, I now sketch how composable contexts might be used to treat not just quantification but also the other linguistic side effects listed in (9) and Table 1. These sketches are by necessity rough and incomplete; they are intended only to convey the general flavor desired—and I believe achievable—for a study of linguistic side effects.

Take INTENSIONALITY. On the prevailing view, even though the morning star and the evening star are both Venus, the phrases *the morning star* and *the evening star* do not have the same denotation. Rather, *the morning star* denotes a function from possible worlds to (possible) things: it maps each possible world  $w$  to the morning star in  $w$ . Similarly, *the evening star* denotes map from each possible world  $w$  to the evening star in  $w$ . Introducing functions from possible worlds in this way allows semanticists to account for the referential

opacity demonstrated in (5) on page 2. We can incorporate the basic idea in our grammar by postulating the following denotations.

$$(60) \quad \llbracket \text{the morning star} \rrbracket = \text{shift } f: \lambda w. f(\text{the morning star in } w)(w) : \text{THING},$$

$$(61) \quad \llbracket \text{the evening star} \rrbracket = \text{shift } f: \lambda w. f(\text{the evening star in } w)(w) : \text{THING}.$$

Denotations for so-called OPAQUE verbs like *think* can then be engineered to account for the empirical facts on (5).

Take VARIABLE BINDING. Back in §1.2, I suggested that pronouns can be viewed as “load instructions”, and antecedents as “store instructions”. To store an antecedent for subsequent reference, we can use the following function from THING to THING. It returns the same THING as it is passed, but incurs a side effect when evaluated.

$$(62) \quad \text{store} = \lambda x. \text{shift } f: f(x)(x) : \text{THING} \rightarrow \text{THING}$$

To load a previously stored object, we can use the following code, which must be evaluated after a store instruction.

$$(63) \quad \text{load} = \text{shift } f: f \quad : \text{THING}$$

Under left-to-right evaluation, these denotations account for the empirical observations in (10) on page 4.

Take INTERROGATIVES. A popular denotational treatment of questions—both direct ones like *Who left?* and indirect ones as in *Alice knows who left*—is to take an interrogative clause to denote the function that maps “questioned objects” to overall propositions (Krifka 2001; *inter alia*). For example, *who left* should denote the function mapping each person  $x$  to the proposition LEFT( $x$ ). We can add *wh*-phrases to our grammar as follows.

$$(64) \quad \llbracket \text{who} \rrbracket = \text{shift } f: \lambda x. \text{ANIMATE}(x) \wedge f(x) : \text{THING}$$

$$(65) \quad \llbracket \text{what} \rrbracket = \text{shift } f: \lambda x. \neg \text{ANIMATE}(x) \wedge f(x) : \text{THING}$$

Just as with intensionality, variable binding, quantification, and interrogatives, simple analyses for FOCUS and PRESUPPOSITIONS can be implemented using *shift* and *reset*. The encodings are straightforward but call for product types (for focus) and sum types (for presuppositions); I omit them here. Suffice it to say that, because the type-lifting operations encountered by linguists are often the same ones used by computer scientists (as Table 1 shows), Filinski’s (1996) general method for encoding monadic effects using composable contexts immediate applies to guide us in specifying denotations like those above.

**6.2. Interaction.** Linguistic research regularly involves multiple phenomena occurring together. For example, studies on questions often consider interrogative determiners, like *which*, alongside quantificational ones, like *every*. Just as a treatment of computational side effects should let them be combined, a uniform theory of linguistic side effects should account for interactions among them. The same tools and techniques may be applicable in both arenas. For example, higher-order continuations have been used to combine both computational side effects (Danvy and Filinski 1990; Filinski 1996) and linguistic ones (Barker 2000; Shan 2002; Shan and Barker 2003).

As a matter of engineering practice, programming language designers aim for features to interact with each other as coherent, self-contained modules. Given several programming language features and a semantics for each, they want to incorporate all of them into a single programming language. In particular, they want to mix and match computational side effects at will, but such an ideal has yet to be achieved. I speculate that the difficulties

computer scientists encounter in combining side effects correspond to restrictions linguists observe on how side effects interact.

**6.3. Evaluation.** The concepts of EVALUATION ORDER and PARAMETER PASSING are crucial to programming languages. Roughly speaking, evaluation order specifies *when* to evaluate each program expression, and parameter passing determines *which* expressions to evaluate. Evaluation order is closely related to side effects: in a programming language without any side effect, neither evaluation order nor choice of parameter-passing mechanism makes any observable difference.

I hypothesize that evaluation order and parameter passing can help account for a variety of generalizations in linguistics. In particular, CROSSOVER and SUPERIORITY phenomena are unified by a notion of left-to-right evaluation (Shan and Barker 2003). More speculatively, the distinction between call-by-value and call-by-name parameter passing (Plotkin 1975) may underlie DE RE VERSUS DE DICTO READINGS AS WELL AS SLOPPY VERSUS STRICT IDENTITY.

**6.4. Summary.** Uniformity, interaction, and evaluation have fruitfully guided programming language research, and I plan to transfer them to linguistics. I aim to develop a theory of linguistic side effects whose uniformity (between interrogatives and quantification, say) simplifies our linguistic theories, points to new generalizations on the ways linguistic side effects interact with each other, and relates denotational (“static”) views of language to operational (“dynamic”) ones.

#### REFERENCES

- Barker, Chris. 2000. Notes on higher-order continuations. Manuscript, University of California, San Diego.
- . 2001. Continuations: In-situ quantification without storage or type-shifting. In *SALT XI: Semantics and linguistic theory*, ed. Rachel Hastings, Brendan Jackson, and Zsolt Zvolensky. Ithaca: Cornell University Press.
- . 2002. Continuations and the nature of quantification. *Natural Language Semantics*. To appear.
- Barwise, Jon, and Robin Cooper. 1981. Generalized quantifiers and natural language. *Linguistics and Philosophy* 4:159–219.
- Carpenter, Bob. 1997. *Type-logical semantics*. Cambridge: MIT Press.
- Danvy, Olivier, and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.
- Filinski, Andrzej. 1996. Controlling effects. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-96-119.
- de Groote, Philippe. 2001. Type raising, continuations, and classical logic. In van Rooy and Stokhof (2001), 97–101.
- Hamblin, C. L. 1973. Questions in Montague English. *Foundations of Language* 10:41–53.
- Hatcliff, John, and Olivier Danvy. 1997. Thunks and the  $\lambda$ -calculus. *Journal of Functional Programming* 7(3):303–319.
- Hendriks, Herman. 1993. Studied flexibility: Categories and types in syntax and semantics. Ph.D. thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam.
- Hobbs, Jerry R., and Stuart M. Shieber. 1987. An algorithm for generating quantifier scopings. *Computational Linguistics* 13(1–2):47–63.

- Jacobson, Pauline. 1999. Towards a variable-free semantics. *Linguistics and Philosophy* 22(2):117–184.
- Krifka, Manfred. 2001. For a structured meaning account of questions and answers. In *Audiatu vox sapientiae: A festschrift for Arnim von Stechow*, ed. Caroline Féry and Wolfgang Sternefeld, 287–319. Berlin: Akademie Verlag.
- Lewin, Ian. 1990. A quantifier scoping algorithm without a free variable constraint. In *COLING '90: Proceedings of the 13th international conference on computational linguistics*, vol. 3, 190–194.
- May, Robert. 1985. *Logical form: Its structure and derivation*. Cambridge: MIT Press.
- Moggi, Eugenio. 1991. Notions of computation and monads. *Information and Computation* 93(1):55–92.
- Montague, Richard. 1974. The proper treatment of quantification in ordinary English. In *Formal philosophy: Selected papers of Richard Montague*, ed. Richmond Thomason, 247–270. New Haven: Yale University Press.
- Moran, Douglas B. 1988. Quantifier scoping in the SRI core language engine. In *Proceedings of the 26th annual meeting of the Association for Computational Linguistics*, 33–40. Somerset, NJ: Association for Computational Linguistics.
- Plotkin, Gordon D. 1975. Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science* 1(2):125–159.
- Quine, Willard Van Orman. 1960. *Word and object*. Cambridge: MIT Press.
- Rooth, Mats Edward. 1985. Association with focus. Ph.D. thesis, Department of Linguistics, University of Massachusetts.
- van Rooy, Robert, and Martin Stokhof, eds. 2001. *Proceedings of the 13th Amsterdam Colloquium*. Institute for Logic, Language and Computation, Universiteit van Amsterdam.
- Shan, Chung-chieh. 2001a. Monads for natural language semantics. In *Proceedings of the ESSLLI-2001 student session*, ed. Kristina Striegnitz, 285–298. Helsinki: 13th European Summer School in Logic, Language and Information.
- . 2001b. A variable-free dynamic semantics. In van Rooy and Stokhof (2001), 204–209.
- . 2002. A continuation semantics of interrogatives that accounts for Baker's ambiguity. In *SALT XII: Semantics and linguistic theory*, ed. Brendan Jackson, 246–265. Ithaca: Cornell University Press.
- . 2003. Quantifier strengths predict scopal possibilities of Mandarin Chinese *wh*-indefinites. Draft manuscript, Harvard University; <http://www.eecs.harvard.edu/~ccshan/mandarin/>.
- Shan, Chung-chieh, and Chris Barker. 2003. Explaining crossover and superiority as left-to-right evaluation. Draft manuscript, Harvard University and University of California, San Diego; <http://semanticsarchive.net/Archive/TBjZDQ3Z/>.
- Spivey, J. Michael. 1990. A functional theory of exceptions. *Science of Computer Programming* 14(1):25–42.
- Wadler, Philip. 1995. Monads for functional programming. In *Advanced functional programming: First international spring school on advanced functional programming techniques*, ed. Johan Jeuring and Erik Meijer, 24–52. Lecture Notes in Computer Science 925, Berlin: Springer-Verlag.