LINGUISTIC SIDE EFFECTS

CHUNG-CHIEH SHAN

This paper relates cases of *apparent noncompositionality* in natural languages to those in programming languages. It is shaped like an hourglass: I begin in §1 with an approach to the syntax-semantics interface that helps us build compositional semantic theories. That approach is to draw an analogy between *computational side effects* in programming languages and what I term by analogy *linguistic side effects* in natural languages.

> This connection can benefit computer scientists as well as linguists, but I focus here on the latter direction of technology transfer. *Continuations* have been useful for treating computational side effects. In §2, I introduce a new metalanguage for continuations in semantics.
>
> > The metalanguage I introduce is useful for analyzing both programming languages and natural languages. For intuition, I survey the first use in §3, then point out the virtues of this treatment in §4.
>
> Turning to natural language in §5, I describe in detail how this perspective helped Chris Barker and I study binding and crossover, as well as *wh*-questions and superiority. I have also used continuations to study quantifier and *wh*-indefinite scope, particularly in Mandarin Chinese, but there is only room here to sketch these further developments, in §6.

Underlying this work are three themes, to be explicated below:

- *uniformity* across side effects,
- *interaction* among side effects, and
- an operational notion of *evaluation*.

To conclude, I will speculatively elevate these methodological preferences into empirical claims in §7.

1. Side effects

Let me begin with Frege's painfully familiar example (Frege 1891, 1892; Quine 1960).

(1) a. the morning star
    b. the evening star

This is a final draft of March 5, 2005 (revision 607). Thanks to Stuart Shieber, Karlos Arregi, Chris Barker, Daniel Büring, Matthias Felleisen, Andrzej Filinski, Danny Fox, Barbara Grosz, Daniel Hardt, C.-T. James Huang, Sabine Iatridou, Pauline Jacobson, Aravind Joshi, Jo-wang Lin, Fernando Pereira, Avi Pfeffer, Chris Potts, Norman Ramsey, Dylan Thurston, Yoad Winter, and the audiences at Harvard AI Research Group, OGI Pacific Software Research Center, Boston University Church Research Group, 8th New England Programming Languages and Systems Symposium, 2003 Harvard Linguistics Practicum in Syntax and Phonology, University of Vermont, and University of Pennsylvania. This research is supported by the United States National Science Foundation Grants IRI-9712068 and BCS-0236592.

The semantic problem with these two phrases starts with the pretheoretic intuition that they "mean" the same thing, namely Venus. Furthermore, we have the intuition that the "meaning" of a sentence like

(2) Alice saw the morning star

is (or at least includes) whether it is true or false. Perhaps we have all been brainwashed the same way in our introductory semantics courses. In any case, these pretheoretic intuitions we have about what phrases mean are at odds with compositionality, because there exist contexts, involving words like *think*, under which *the morning star* and *the evening star* are no longer interchangeable: maybe Alice thinks Bill saw the morning star, but Alice doesn't think Bill saw the evening star.

In this example, we have some pretheoretic notions of what certain phrases mean, which turn out to be incompatible with compositionality. In other cases, we are uncertain what certain phrases should mean at all, for example *the king of France (is bald)* or *most unicorns (are happy)*. That's when we are tempted to concoct technical devices like partial functions or syntactic movement, so that a larger phrase may have a meaning without each of its constituent parts also having a meaning.

The same challenge faces programming languages (Søndergaard and Sestoft 1990, 1992). For example, here are two program phrases (which happen to be in the Python programming language, but most of them look alike anyway).

(3) a. $f(2) \times f(3)$
b. $f(3) \times f(2)$

The first program means to apply the function f to the numbers 2 and 3, then multiply the results. The second program means to apply the function f to the numbers 3 and 2, then multiply the results. These two programs intuitively have the same meaning, but as it turns out, there too are contexts that distinguish between them. For example, suppose that we define the function f to print out its argument to the terminal before returning it.

(4)
```
def f(x):
    print x × 10
    return x
```

Then $f(2) \times f(3)$ would print out "20 30" before returning 6, while $f(3) \times f(2)$ would print out "30 20" before also returning 6. We can blame this discrepancy on the presence in our programming language of a command like "print". And this is only a mild case: what denotations, if any, can we assign to program phrases that request input from the user, or commands like "goto" that jump to a different point in the program to continue executing?

The commonality here between natural and programming language semantics is that we have some pretheoretic notion of what phrases mean—for example, the expression $f(2)$ "means" whatever you get when you feed the function f the number 2, and the noun phrase *the morning star* "means" Venus—yet two phrases that supposedly mean the same thing turn out to be distinguished by a troublemaking context involving verbs like *think* or commands like "print". This kind of situation—where, in short, equals cannot be substituted for equals—is what I take *referential opacity* to mean (as opposed to *referential transparency*, which is when equals can be substituted for equals). A vaguer way to define referential opacity is that it's when meaning "depends on context". Worse than referential opacity, sometimes we don't have any pretheoretic notion of meaning. For example, what does *the king of France* mean, and what does "goto" mean, anyway?

These issues, of referential opacity and downright lack of reference, are common to both programming and natural languages. Programming language researchers call instances of these issues *computational side effects*, such as

- output ("print"),
- input ("read"), and
- control ("goto").

By analogy, I call a natural language phenomenon a *linguistic side effect* when it involves either referential opacity or the lack of any pretheoretic notion of meaning to even challenge compositionality with. Some examples of linguistic side effects are:

- intensionality (*think*),
- binding (*she*),
- quantification (*most*),
- interrogatives (*who*),
- focus (*only*), and
- presuppositions (*the king of France*).

As I said, side effects are a common problem in both natural and programming language semantics. A way to treat side effects that is very popular in both linguistics and computer science is *type-lifting*—in other words, enriching denotations to encompass the additional aspect of "meaning" under discussion. For example, in order to distinguish denotationally between *the morning star* and *the evening star*, it is standard to intensionalize a natural language semantics, introducing functions from possible worlds. For another example, programming language semanticists deal with "print" by lifting denotations from numbers to number-string pairs, where the string is the terminal output.

Just to be complete, I should acknowledge that lifting denotation types is not the whole story. Whenever we lift denotations, we also have to lift the composition rules that combine them. Moreover, we need to specify how to get a truth value at the top level from a semantic value that is now richer, more complicated.

The type lifting operations used in linguistics and in computer science are very similar. I mentioned above the standard possible world semantics for intensionality. The same idea of turning denotations into functions from a fixed set is used to treat the computational side effect of input in programming languages. A second case where linguistics and computer science came up with the same type-lifting is Hamblin's alternative semantics for questions (1973), which is how *nondeterminism* in programming languages—commands like "flip a coin" or "roll a die"—are often analyzed.

A third case of this convergence in type-lifting is quantification (Montague 1974). It is the focus of the rest of this paper. As linguists, we know how useful generalized quantifiers are, but I want to tell you how computer scientists use them to model programming languages, then come back after a few sections to apply their perspective to linguistics.

2. A NEW METALANGUAGE

To ground the discussion, I now introduce a toy programming language, where everything is a string. The + sign means to concatenate two strings, so the program

(5) "compositional" + " " + "semantics"

concatenates the strings "compositional", a space, and "semantics". When a computer executes this program, it first concatenates "compositional" and a space, and then concatenates the result with "semantics". The result is "compositional semantics". I write this

computation as a sequence of reductions:

(6)
$$\begin{aligned} &\underline{\text{“compositional” + “ ”}} + \text{“semantics”} \\ &\quad\Rightarrow\ \underline{\text{“compositional ” + “semantics”}} \\ &\quad\Rightarrow\ \text{“compositional semantics”} \end{aligned}$$

Each $\Rightarrow$ indicates a step, and underlining indicates the subexpression next reduced. So far everything is pretty straightforward, and it is easy to write down a trivial denotational semantics for this language, where quotations denote strings, the + sign denotes the string concatenation function, and so on.

Now I add two features to this programming language, due to Danvy and Filinski (1989, 1990, 1992), with roots in work by Felleisen (1987, 1988) and others. These features may seem weird at first, but bear with me for the moment. The *shift* command is written $\xi f.\, e$, where f is a variable name and e is an expression. The symbol ξ in a shift-expression plays the same role as λ does in a function expression: it opens the scope of a variable-binding construct. When $\xi f.\, e$ is reduced, it first sets aside its context as f. For example, in the program

(7)
$$\text{“compositional ” + } (\underline{\xi f.\, \text{“directly ” + } f(\text{“semantics”})}),$$

the context of the shift-expression is

(8)
$$\text{“compositional ” + } \_\_,$$

where the blank $\_\_$ indicates the location of the shift-expression itself. This context is bound to the variable f within the body e. Furthermore, the body e (in this example, “directly ” + f(“semantics”)) becomes the new current expression to be evaluated. Hence the following reduction sequence. (Ignore the square brackets for now; we shall come to them momentarily.)

(9)
$$\begin{aligned} &\text{“compositional ” + } (\underline{\xi f.\, \text{“directly ” + } f(\text{“semantics”})}) \\ &\quad\Rightarrow\ \text{“directly ” + } \underline{(\lambda x.\, [\text{“compositional ” + } x])(\text{“semantics”})} \\ &\quad\Rightarrow\ \text{“directly ” + } \underline{[\text{“compositional ” + “semantics”}]} \\ &\quad\Rightarrow\ \text{“directly ” + } \underline{[\text{“compositional semantics”}]} \\ &\quad\Rightarrow\ \underline{\text{“directly ” + “compositional semantics”}} \\ &\quad\Rightarrow\ \text{“directly compositional semantics”} \end{aligned}$$

The result of the very first reduction is the body of the shift-expression, with f replaced by $\lambda x.\, [\text{“compositional ” + } x]$, which is a function that represents the context (8).

The second feature to add to this programming language is *reset*. A reset is notated by a pair of square brackets, like those in (9). Resets delimit the extent to which an enclosed shift can grab its surrounding context. For example, in the program

(10)
$$\text{“really ” + } [\text{“compositional ” + } (\xi f.\, \text{“directly ” + } f(\text{“semantics”}))]\,,$$

the shift can only grab its surrounding context as far as (and excluding) the closest enclosing square brackets, which shield “really ” from being captured. Hence “really ” stays at

the beginning of the program throughout the reduction sequence.

(11) “really ” + [“compositional ” + (ξf. “directly ” + f(“semantics”))]
⇒ “really ” + [“directly ” + (λx. [“compositional ” + x])(“semantics”)]
⇒ “really ” + [“directly ” + [“compositional ” + “semantics”]]
⇒ “really ” + [“directly ” + [“compositional semantics”]]
⇒ “really ” + [“directly ” + “compositional semantics”]
⇒ “really ” + [“directly compositional semantics”]
⇒ “really ” + “directly compositional semantics”
⇒ “really directly compositional semantics”

We can imagine that every program is implicitly surrounded by a reset, so in the absence of any explicit bracketing, shift grabs all of its surrounding context.

As a technical detail (Shan 2004), when shift grabs a context and stores it in a bound variable, the stored context is bracketed by reset. That's why the context (8) corresponds to the function λx. [“compositional ” + x], not just λx. (“compositional ” + x).

Another technical detail: If you are used to working with the pure λ-calculus, you might have the habit of looking for whatever subexpression can be reduced by λ-conversion and reducing it right away. It used to be that different reduction orders always give the same final result, but that is no longer so in the presence of shift and reset. Instead, we need to replace that habit with a systematic recursive traversal that turns each subexpression (like concatenation) into a value (like a string) before moving on.[1] In particular, no reduction is allowed in the body of a λ-abstraction. Moreover, λ-conversion can take place only if the argument is an (irreducible) value. In computer science parlance, this programming language *passes parameters by value*, or is *call-by-value*. Call-by-value is not the only parameter-passing convention, but it is the most popular one among programming languages, and the only one for which shift and reset have been defined and studied.

3. Encoding computational side effects

Shift and reset, like “goto”, are known as *control operators* in computer science. Unlike “goto”, shift and reset are control operators that manipulate *delimited contexts*,[2] which means that the context grabbed by shift is made available to the program in the form of a function (Felleisen 1987, 1988). Shift and reset are interesting to computer scientists because many computational side effects can be encoded with them, in other words treated as syntactic sugar for them (Filinski 1994, 1996, 1999). I give four examples below.

3.1. **Abort.**

A popular feature of programming languages is to be able to abort a computation in the middle of its execution (to “throw an exception”). To model such a feature, we might want an “abort” command, such that

(12) “directly ” + abort(“compositional”) + “ semantics”

evaluates to just “compositional”. To achieve this, we can treat abort as simply a special case of shift, where the shifted context f is never used.

(13) abort = $\lambda x. \xi f. x$

[1] Or better, use Kameyama and Hasegawa's (2003) axiomatization of shift and reset. Their equations are sound and complete with respect to observational equivalence under reductions by the systematic recursive traversal described here.

[2] Delimited contexts are also known as *composable*, *partial*, *functional*, or *truncated* contexts.

Substituting (13) into (12) gives the following reduction sequence, as desired.

(14) “directly ” + $\underline{(\lambda x.\, \xi f.\, x)(\text{“compositional”})}$ + “ semantics”
$\Rightarrow$ “directly ” + $(\underline{\xi f.\, \text{“compositional”}})$ + “ semantics”
$\Rightarrow$ “compositional”

3.2. **Random.** Another popular feature, which I alluded to above, is nondeterminism. We want the program

(15) random(“direct”)(“indirect”) + “ly”

to evaluate to a set containing two strings, “directly” and “indirectly”. We can treat random as another special case of shift, as long as we turn the overall expression into a singleton set (hence the outermost pair of braces in (17)).

(16) $\text{random} = \lambda x.\, \lambda y.\, \xi f.\, f(x) \cup f(y)$

(17) $\{\underline{(\lambda x.\, \lambda y.\, \xi f.\, f(x) \cup f(y))(\text{“direct”})}(\text{“indirect”}) + \text{“ly”}\}$
$\Rightarrow \{\underline{(\lambda y.\, \xi f.\, f(\text{“direct”}) \cup f(y))(\text{“indirect”})} + \text{“ly”}\}$
$\Rightarrow \{\underline{\xi f.\, f(\text{“direct”}) \cup f(\text{“indirect”})} + \text{“ly”}\}$
$\Rightarrow \underline{(\lambda x.\, [\{x + \text{“ly”}\}])(\text{“direct”})} \cup (\lambda x.\, [\{x + \text{“ly”}\}])(\text{“indirect”})$
$\Rightarrow [\{\underline{\text{“direct”} + \text{“ly”}}\}] \cup (\lambda x.\, [\{x + \text{“ly”}\}])(\text{“indirect”})$
$\Rightarrow \underline{[\{\text{“directly”}\}]} \cup (\lambda x.\, [\{x + \text{“ly”}\}])(\text{“indirect”})$
$\Rightarrow \{\text{“directly”}\} \cup \underline{(\lambda x.\, [\{x + \text{“ly”}\}])(\text{“indirect”})}$
$\Rightarrow \{\text{“directly”}\} \cup [\{\underline{\text{“indirect”} + \text{“ly”}}\}]$
$\Rightarrow \{\text{“directly”}\} \cup \underline{[\{\text{“indirectly”}\}]}$
$\Rightarrow \underline{\{\text{“directly”}\} \cup \{\text{“indirectly”}\}}$
$\Rightarrow \{\text{“directly”}, \text{“indirectly”}\}$

The two computational side effects discussed so far, abort and random, are encoded by shift expressions whose bodies use the captured context f either never (for abort) or twice (for random). By contrast, the computational and linguistic side effects considered below are encoded by shift expressions that use the captured context exactly once.

3.3. **Input.** A third important feature of programming languages, which I also mentioned above, is input. We want the program

(18) input + “ semantics”

to evaluate to the function that appends the word “semantics” to every string. We can write input in terms of shift as well. It is just $\xi f.\, f$.

(19) $\text{input} = \xi f.\, f$

(20) $\underline{(\xi f.\, f)} + \text{“ semantics”} \quad \Rightarrow \lambda x.\, [x + \text{“ semantics”}]$

3.4. **Output.** We can treat output in the same framework as input. We want the program

(21) output(“semantics”) + “ ” + input

to evaluate to “semantics semantics”, where the second word “semantics” is fed to the input-expression to the right by the output-expression to the left. Once again, output can

be written in terms of shift.[3]

(22) $$\text{output} = \lambda x.\, \xi f.\, f(x)(x)$$

(23)
$$\begin{aligned}
&\underline{(\lambda x.\, \xi f.\, f(x)(x))(\text{“semantics”})} + \text{“ ”} + (\xi f.\, f)\\
&\quad\Rightarrow\ (\underline{\xi f.\, f(\text{“semantics”})(\text{“semantics”})}) + \text{“ ”} + (\xi f.\, f)\\
&\quad\Rightarrow\ \underline{(\lambda y.\, [y + \text{“ ”} + (\xi f.\, f)])(\text{“semantics”})}(\text{“semantics”})\\
&\quad\Rightarrow\ [\underline{\text{“semantics”} + \text{“ ”}} + (\xi f.\, f)]\,(\text{“semantics”})\\
&\quad\Rightarrow\ [\text{“semantics ”} + (\underline{\xi f.\, f})]\,(\text{“semantics”})\\
&\quad\Rightarrow\ \underline{[\lambda z.\, [\text{“semantics ”} + z]]}(\text{“semantics”})\\
&\quad\Rightarrow\ \underline{(\lambda z.\, [\text{“semantics ”} + z])(\text{“semantics”})}\\
&\quad\Rightarrow\ \underline{[\text{“semantics ”} + \text{“semantics”}]}\\
&\quad\Rightarrow\ \underline{[\text{“semantics semantics”}]}\\
&\quad\Rightarrow\ \text{“semantics semantics”}
\end{aligned}$$

You may be able to see where I am going: output is like the creation of a discourse referent, and input is like a pronoun to be bound. Indeed, that is what relates shift and reset to dynamic semantics, not to mention crossover in binding and superiority in *wh*-questions. However, before I move back to linguistics, let me point out three crucial virtues of shift and reset.

4. Three virtues of shift and reset

The first virtue of shift and reset is that there is a perfectly compositional denotational semantics for them.

So far I have described shift and reset in terms of how they are evaluated, which gives them a very operational—or, in linguistic terms, derivational—feel. But thanks to this denotational semantics, we can treat this programming language with shift and reset as just another metalanguage for writing down model-theoretic denotations: good old functions and sets and so on in the simply typed λ-calculus. The only theoretical commitment that is required of us before we can use shift and reset in our semantic theories is that there are modes of semantic combination other than pure function application (like Montague's use of variables (1974) and Hamblin's of alternatives (1973)). Not even type-shifting is needed if you don't like that. I won't go into details about this denotational semantics here, but it is based on *continuations*, which generalize generalized quantifiers (Barker 2002; Shan 2002). Thus even computer scientists with nothing to do with natural language still care about generalized quantifiers, though many of them have never heard of the term.

The translation from a metalanguage with shift and reset to one without is called the *continuation-passing-style* (CPS) transform.[4] The type system of the CPS transform's target language serves as a refined type system for the source language with shift and reset. For example, the expressions for input and output in (19) and (22) above may look like they have the types e and $\langle e, e\rangle$, respectively, but they translate to λ-terms with the types

(24) $$\langle\langle \boxed{e}, \alpha\rangle, \langle e, \alpha\rangle\rangle$$

[3]This encoding of input and output forces invocations of input and output to match one-to-one. For applications such as sloppy identity in verb-phrase ellipsis, this requirement may not be desirable. If so, we can instead define input $= \xi f.\, \xi x.\, f(x)(x)$ and apply the function $\lambda v.\, \lambda x.\, v$ at the top level of all programs. This way, each output can feed zero or more inputs.

[4]As Danvy and Filinski (1990) point out, the translation for shift and reset should technically be called the continuation-*composing*-style transform instead.

and

$$\langle\langle \boxed{\langle e,} \langle\langle \boxed{e} , \langle e, \gamma\rangle\rangle, \gamma\rangle \boxed{\rangle} , \beta\rangle, \beta\rangle, \tag{25}$$

respectively. Here α, β, and γ are type variables that can be instantiated with any type. The boxed portions in the types above are vestiges of the pre-refinement types e and $\langle e, e\rangle$.

The second virtue of shift and reset is that they connect our desire for a denotational semantics that specifies what each phrase means to our intuitive understanding of what happens when a computer executes a program or when a person processes a sentence.

For example, in the Python example (3) at the beginning of this paper, it is intuitive that if $f(2)$ is evaluated before $f(3)$, then 2 is printed before 3, and vice versa. This notion of *evaluation order* is preserved in our treatment of output in terms of shift and reset. That is, we can take "print" to be shorthand for a shift expression, such that, if the shift expression for "print 2" is evaluated before that for "print 3", then 2 is printed before 3, and vice versa.

Similarly, recall from §3 that the program (21), repeated below, produces the result "semantics semantics".

$$\text{output}(\text{“semantics”}) + \text{“ ”} + \text{input} \tag{21}$$

For the input above to successfully consume the output, the shift-expression for output must be reduced (evaluated) before that for input. In particular, if we stipulate that evaluation takes place from left to right, then the "flipped" program

$$\text{input} + \text{“ ”} + \text{output}(\text{“semantics”}) \tag{26}$$

won't work:

$$\begin{aligned} &\underline{(\xi f.\, f)} + \text{“ ”} + (\lambda x.\, \xi f.\, f(x)(x))(\text{“semantics”}) \\ &\quad \Rightarrow\ \lambda x.\, [x + \text{“ ”} + (\lambda x.\, \xi f.\, f(x)(x))(\text{“semantics”})] \end{aligned} \tag{27}$$

Evaluation stops after one reduction, resulting in a function that waits for an input x—an input that the yet-to-be-evaluated output of "semantics" fails to provide. Even if we provide some input (say "syntax") "by hand", evaluation still halts at an attempt to apply a string as a function to another string:

$$\begin{aligned} &(\lambda x.\, [x + \text{“ ”} + (\lambda x.\, \xi f.\, f(x)(x))(\text{“semantics”})])(\text{“syntax”}) \\ &\quad \Rightarrow\ [\underline{\text{“syntax”} + \text{“ ”}} + (\lambda x.\, \xi f.\, f(x)(x))(\text{“semantics”})] \\ &\quad \Rightarrow\ [\text{“syntax ”} + \underline{(\lambda x.\, \xi f.\, f(x)(x))(\text{“semantics”})}] \\ &\quad \Rightarrow\ [\text{“syntax ”} + (\underline{\xi f.\, f(\text{“semantics”})(\text{“semantics”})})] \\ &\quad \Rightarrow\ [\underline{(\lambda x.\, [\text{“syntax ”} + x])(\text{“semantics”})}(\text{“semantics”})] \\ &\quad \Rightarrow\ [\,\underline{[\text{“syntax ”} + \text{“semantics”}]}\,(\text{“semantics”})] \\ &\quad \Rightarrow\ [\underline{[\text{“syntax semantics”}]}(\text{“semantics”})] \\ &\quad \Rightarrow\ [\underline{\text{“syntax semantics”}(\text{“semantics”})}] \\ &\quad \Rightarrow\ \text{type error!} \end{aligned} \tag{28}$$

The refined type system mentioned above flags this problem as a type error in (26).

The shift-reset metalanguage thus provides a link between our operational impulses and our denotational desires. Specifically, continuations provide a denotational foundation for the operational notion of evaluation order. There are other denotational foundations, such as Moggi's computational metalanguage (1991), that are less concrete than continuations.

The third virtue of shift and reset is that we don't just know how to treat *many* computational side effects in terms of shift and reset, in other words how to translate abort,

random, input, output, etc. into shift and reset. There turns out to be a systematic procedure for implementing *any* computational side effect in terms of shift and reset, under a certain technical definition of what a computational side effect is. The technical details are in Filinski's doctoral dissertation (1996), but you don't need to read it to get the hang of the procedure. For instance, when treating nondeterminism in §3.2 above, the top-level expression in (17) needs to be a singleton set, which can be thought of as the trivial amount of nondeterminism. That is a step prescribed by Filinski's systematic procedure: at the top level, always put the trivial amount of whatever computational side effect you want to encode.

Because shift and reset are so broadly applicable, they allow computer scientists to treat multiple computational side effects in a uniform framework, and linguists to treat multiple linguistic side effects in a uniform framework. Instead of lifting our semantics once, and differently, for each of intensionality and variable binding and tense and questions and focus and quantification and indefinites and conventional implicature—just think of how complicated the denotation of *John* can get before all these are taken into account, with or without dynamic type-shifting—we need only specify a single lifting of types and denotations, to which everything can be reduced. Or so I hope.

5. Encoding linguistic side effects

Having expressed the hope to treat all linguistic side effects in a uniform framework, I will now get cranking.

5.1. **Quantification.** First on the agenda is in-situ quantification.

(29) Alice loves everyone's mother.

We want the program

(30) $$\text{alice}\backslash(\text{love}/(\text{everyone}\backslash\text{mother}))$$

(forward and backward slashes denote forward and backward function application) to evaluate to $\forall x.\,\text{love}(\text{mother}(x))(\text{alice})$. The standard denotation of *everyone*, as given by Montague (1974), can be translated into our λ-calculus enriched with shift and reset.

(31) $$\text{everyone} = \xi f.\,\forall x.\,f(x).$$

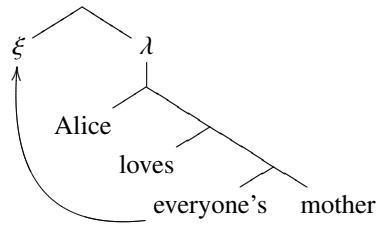
This definition enables the reduction sequence

(32)
$$\begin{aligned}
&\text{alice}\backslash(\text{love}/((\underline{\xi f.\,\forall x.\,f(x)})\backslash\text{mother})) \\
&\quad\Rightarrow\ \forall x.\,\underline{(\lambda y.\,[\text{alice}\backslash(\text{love}/(y\backslash\text{mother}))])(x)} \\
&\quad\Rightarrow\ \forall x.\,[\text{alice}\underline{\backslash}(\text{love}\underline{/}(x\underline{\backslash}\text{mother}))] \\
&\quad\Rightarrow\ \forall x.\,\underline{[\text{love}(\text{mother}(x))(\text{alice})]} \\
&\quad\Rightarrow\ \forall x.\,\text{love}(\text{mother}(x))(\text{alice}),
\end{aligned}$$

as desired.

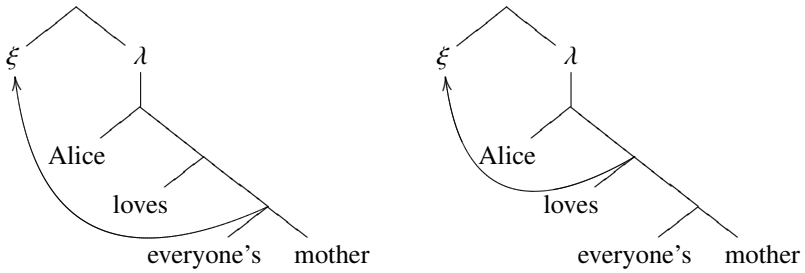
If you think of shift from the reduction (or operational) point of view, you might be reminded of covert movement and predicate abstraction. For example, the reduction sequence above may be depicted as the following tree.

(33)



This is a fine intuition to have as a first approximation, but it is only an approximation. To reiterate, shift and reset are simply notation for denotations in a directly compositional semantics based on continuations. One difference that this compositional nature makes is that constituents containing *everyone*, like *everyone's mother* and *loves everyone's mother*, are every bit as quantificational as *everyone* is.[5] There is no reason other than tradition to draw (32) as (33) rather than either of the following pictures.

(34)



If *X* is a noun phrase that incurs side effects, then *X's mother* incurs side effects as well.[6] This fact will be useful when we treat pied-piping in §5.4 below.

5.2. **Binding.** As I alluded to above, binding in natural language can be decomposed into output (the creation of discourse referents) and input (the evaluation of a pronoun). This analogy to output and input in programming languages (§3.3–4) is the basic idea behind the continuation-based analysis of binding and crossover by Chris Barker and I (Shan and Barker 2004). To take one example, the sentence

(35) Everyone's$_i$ father loves her$_i$ mother

performs output at *everyone* and input at *her*. We would like the corresponding program

(36) (output(everyone)\father)\(love/(input\mother))

[5]This property is shared by other compositional treatments of in-situ quantification, such as Cooper storage (1983), Keller storage (1988), and Moortgat's type constructor *q* (1988, 1995, 1996).

[6]A special case of this statement is that, if *X* contains a pronoun yet to be bound, then so does *X's mother*, as in Jacobson's variable-free semantics for binding (1999, 2000).

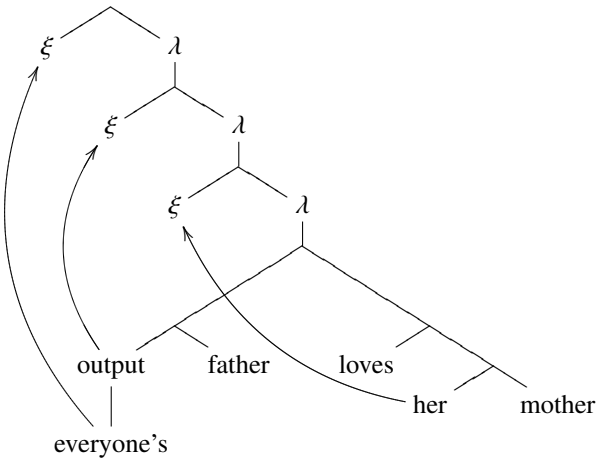
to evaluate to $\forall x.\,\mathrm{love}(\mathrm{mother}(x))(\mathrm{father}(x))$. Luckily, it already does, given the definitions in (19), (22), and (31):

(37)
$$\begin{aligned}
&(\mathrm{output}(\underline{\mathrm{everyone}})\backslash\mathrm{father})\backslash(\mathrm{love}/(\mathrm{input}\backslash\mathrm{mother}))\\
&\Rightarrow\ \forall x.\,\underline{(\lambda y.\,[(\mathrm{output}(y)\backslash\mathrm{father})\backslash(\mathrm{love}/(\mathrm{input}\backslash\mathrm{mother}))])(x)}\\
&\Rightarrow\ \forall x.\,[(\underline{\mathrm{output}(x)}\backslash\mathrm{father})\backslash(\mathrm{love}/(\mathrm{input}\backslash\mathrm{mother}))]\\
&\Rightarrow\ \forall x.\,[((\underline{\xi f.\,f(x)(x)})\backslash\mathrm{father})\backslash(\mathrm{love}/(\mathrm{input}\backslash\mathrm{mother}))]\\
&\Rightarrow\ \forall x.\,[\underline{(\lambda y.\,[(y\backslash\mathrm{father})\backslash(\mathrm{love}/(\mathrm{input}\backslash\mathrm{mother}))])(x)}(x)]\\
&\Rightarrow\ \forall x.\,[\,[(x\backslash\mathrm{father})\backslash(\mathrm{love}/(\underline{\mathrm{input}}\backslash\mathrm{mother}))]\,(x)]\\
&\Rightarrow\ \forall x.\,[\underline{[\lambda y.\,[(x\backslash\mathrm{father})\backslash(\mathrm{love}/(y\backslash\mathrm{mother}))]]}(x)]\\
&\Rightarrow\ \forall x.\,[\underline{(\lambda y.\,[(x\backslash\mathrm{father})\backslash(\mathrm{love}/(y\backslash\mathrm{mother}))])(x)}]\\
&\Rightarrow\ \forall x.\,[\,[(x\underline{\backslash}\mathrm{father})\underline{\backslash}(\mathrm{love}\underline{/}(x\underline{\backslash}\mathrm{mother}))]]\\
&\Rightarrow\ \forall x.\,[\underline{[\mathrm{love}(\mathrm{mother}(x))(\mathrm{father}(x))]}]\\
&\Rightarrow\ \forall x.\,\underline{[\mathrm{love}(\mathrm{mother}(x))(\mathrm{father}(x))]}\\
&\Rightarrow\ \forall x.\,\mathrm{love}(\mathrm{mother}(x))(\mathrm{father}(x))
\end{aligned}$$

Just as our treatment of quantification accounts for quantifiers buried in noun phrases like any other quantifier, this treatment of binding deals with what Büring (2001) calls *binding out of DP* right away. In the sentence above, *everyone*, buried inside *everyone's father*, can still bind *her*.

The reduction sequence above involves three shifts, and so can be approximately depicted by a tree that invokes covert movement and predicate abstraction three times:

(38)



On this view, the binder and the pronoun both raise covertly, such that the landing site of the binder immediately c-commands the landing site of the pronoun. Among the three constituents moved, the linear order of base positions from left to right corresponds to the hierarchical order of landing sites from high to low. Such *tucking-in* (Richards 1997) approximates our stipulation in §4 that expressions be evaluated from left to right. Because *output(everyone)* contains *everyone*, there is no linear precedence to speak of between these two constituents. Rather, *everyone* is an argument to *output*, so the call-by-value evaluation discipline specified in §2 requires that *everyone* be raised first, followed by the remnant *output(__)*, in an instance of inverse linking (May 1977) and remnant movement (den Besten and Webelhuth 1990; inter alia).

Despite these similarities between shift and movement, I hope all these arrows do not obscure the fact that the shift-reset metalanguage is not just a technical implementation of movement and LF. For example, to the extent (45) below can be described in movement terms, it is crucial that reconstruction can take place both before and after Move and Merge, as dictated by the lexical items involved. More complex configurations are possible, especially in cases of ellipsis.

In any case, defaulting to left-to-right evaluation rules out crossover. The unacceptability of

(39) Her$_i$ father loves everyone's$_i$ mother

is predicted, because the program

(40) (input\father)\(love/(output(everyone)\mother))

evaluates input before output, and so gets stuck as (27) on page 8 does. This correct prediction manifests as a type mismatch in the refined type system mentioned in §4. Or, roughly speaking in movement terms, to prohibit crossover is to require crossing rather than nested movement (Shan and Barker 2002).

5.3. **Evaluation order and thunks.** All this talk of left-to-right evaluation and linear order may leave you suspecting that Chris and I predict simply that the binder has to occur to the left of the pronoun. That better not be the case, because of fronted *wh*-phrases in sentences like the following.

(41) a. Whose present for him$_i$ did every boy$_i$ see?
b. *Which boy$_i$ does his$_i$ mother like?

In (41a) is an instance of acceptable cataphora; in (41b) is an instance of unacceptable anaphora.

Fortunately, left-to-right evaluation does not entail that every binder must be pronounced before any pronoun it binds. More generally, evaluation order only affects binary operators like "+" (string concatenation), "\" (backward function application), and "/" (forward function application). Left-to-right evaluation does not mean that one subexpression is evaluated before another whenever the former appears to the left of the latter. To the contrary, call-by-value parameter passing dictates, for instance, that function arguments be fully evaluated before λ-conversion.

For example, in the computer program (42), the command to print "sitional" appears before the command to print "compo". Yet, when the program runs, it prints "compositional", not "sitionalcompo".

(42) def f(): print "sitional"
def g(): print "compo"
g()
f()

Merely defining f to be a function that prints "sitional" does not print "sitional". Merely that the program contains "sitional" before "compo" does not entail that it prints "sitional" before "compo".

In general, the side effects of a piece of code is incurred when it is executed (zero or more times), not where it appears literally in the program text (once). The time of execution can differ from the location of definition, because a program can pass around functions with side effects, like f and g, without invoking them. Hence left-to-right evaluation does not mean that side effects all happen in the order in which they are mentioned in the program.

All it means is that, for an operator like "+", where in principle either branch can be evaluated first, we break the symmetry and evaluate the left branch first.

At this point, it is convenient to introduce a new type into our λ-calculus. The new type is the *singleton* type—a set with only one element in it; it doesn't matter which. The singleton can be thought of as a 0-tuple, just as an ordered pair is a 2-tuple. I write this new type as "()". I also write the unique element of this type as "()". Thus the unique function from the singleton type to the singleton type is $\lambda().\,()$; in fact, there is a unique function from any type to the singleton type, namely $\lambda x.\,()$. The singleton type is also known as the *unit* or *terminal* type in computer science.

In the standard λ-calculus, the singleton type is not useful. For example, a function from () to e is equivalent to just an individual. More generally, the types $\langle (), \alpha \rangle$ and α are always equivalent: every time you see a function whose domain is the singleton type, you may as well strip off that function layer without losing any information.

In the presence of side effects, however, things get more interesting. A function from () to e is no longer necessarily just a function that always returns the same individual; it could be a function that first triggers some side effects (like producing some output or consuming some input), then returns an individual—maybe not even the same one each time! The singleton is just a dummy argument to build a function—an excuse to defer side effects for later, as in (42). Such a function, whose domain is the singleton type, is called a *thunk* (Hatcliff and Danvy 1997; Ingerman 1961).[7]

5.4. ***Wh*-questions; topicalization.** Back to linguistics: How do thunks help us understand the following interactions of quantificational binding with *wh*-questions and topicalization?

(43) a. [Whose present for him$_i$] did every boy$_i$ see?
b. *[Which boy$_i$] does his$_i$ mother like?

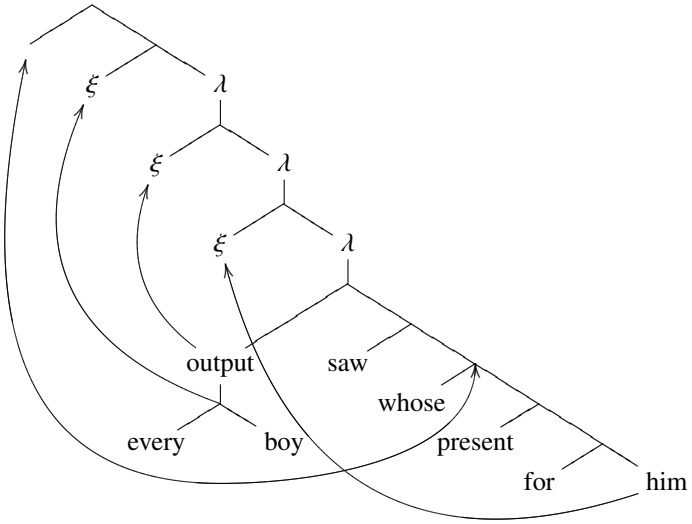
(44) a. [Alice's present for him$_i$], every boy$_i$ saw.
b. *[Every boy$_i$], his$_i$ mother likes.

Fronted phrases, like the bracketed parts of (43–44), may be thought to obligatorily reconstruct to the gap location. The following tree for (43a) indicates this approximate

[7]Etymology: The word "thunk" is "the past tense of 'think' at two in the morning" (Raymond et al. 2003).

understanding by arrows on both ends of the outermost curve.

(45)



First, the quantifier *every boy* raises; second, the remnant *output* tucks in; third, the *wh*-phrase *whose present for him* reconstructs; finally, the pronoun *him* tucks in further.

As one might expect, the crucial piece of this puzzle is how to analyze filler-gap dependencies. We introduce into our grammar a silent element, which I follow tradition in notating as __ and pronouncing as "trace". The trace denotes the following. It should be reminiscent of input in (19).

(46) $$\_\_ = \xi f.\, \lambda g.\, f(g())$$

The best way to understand this denotation is to watch it in action. We can now generate a gapped clause like *Alice loves* __.

(47) $$\begin{aligned} &\text{alice}\backslash(\text{love}/\underline{\_\_}) \\ &\quad\Rightarrow\ \lambda g.\, [(\lambda x.\, \text{alice}\backslash(\text{love}/x))(g())] \end{aligned}$$

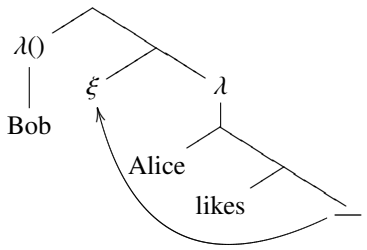
The denotation comes out to be a λ-abstraction of type $\langle\langle(), e\rangle, t\rangle$, in that we need to feed it a thunked individual (in other words, a value of type $\langle(), e\rangle$) in order to recover a saturated proposition. For example, for a sentence like *Bob, Alice loves*, we feed it the thunked individual that simply returns Bob with no side effect.

(48) $$\begin{aligned} &\underline{(\lambda().\, \text{bob})\backslash(\lambda g.\, [(\lambda x.\, \text{alice}\backslash(\text{love}/x))(g())])} \\ &\quad\Rightarrow\ [(\lambda x.\, \text{alice}\backslash(\text{love}/x))(\underline{(\lambda().\, \text{bob})()})] \\ &\quad\Rightarrow\ [\underline{(\lambda x.\, \text{alice}\backslash(\text{love}/x))(\text{bob})}] \\ &\quad\Rightarrow\ [\text{alice}\underline{\backslash}(\text{love}\underline{/}\text{bob})] \\ &\quad\Rightarrow\ \underline{[\text{love}(\text{bob})(\text{alice})]} \\ &\quad\Rightarrow\ \text{love}(\text{bob})(\text{alice}) \end{aligned}$$

One way to think of the trace, as defined here, is as follows. Shift and reset are a concrete implementation of covert movement, but our system has no direct correlate of overt movement. Roughly speaking, our trace encodes overt movement using covert movement: instead of overtly raising some phrase, we raise a silent phrase, the trace. The overt material is base-generated, so to speak, right next to the landing site of this movement. Because the trace is silent, it makes no empirical difference whether it raises covertly or overtly;

our account of fronting claims the denotational equivalent of the trace's moving covertly. Cinque (1990) and Postal (1998) have made similar proposals for overt movement, at least for topicalization.

(49)



In the trivial example (48), the only side effect is the shift by the trace. There is no binding, no quantification, and no *wh*-question. There are a variety of ways in which life can become more exciting. First of all, note that we now have a unified analysis of raised and in-situ *wh*-phrases. Suppose that we analyze a *wh*-phrase like *who* as simply input.

(50) $$\text{who} = \xi f.\, f$$

Then, we can not only treat in-situ-*wh* questions like (51), as shown in (52)—

(51) Alice saw who?

(52) $$\text{alice}\backslash(\text{see}/\underline{\text{who}}) \;\Rightarrow\; \lambda x.\,[\text{alice}\backslash(\text{see}/x)]$$

—but also treat raised-*wh* questions like (53), by thunking the *wh*-phrase and using it in conjunction with the trace:

(53) Who did Alice see?

(54) $$\begin{aligned}
&(\lambda().\,\text{who})\backslash[\text{alice}\backslash(\text{see}/\underline{\quad})] \\
&\quad\Rightarrow\; (\lambda().\,\text{who})\backslash\underline{[\lambda g.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(g())]]} \\
&\quad\Rightarrow\; \underline{(\lambda().\,\text{who})\backslash(\lambda g.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(g())])} \\
&\quad\Rightarrow\; [(\lambda x.\,\text{alice}\backslash(\text{see}/x))(\underline{(\lambda().\,\text{who})()})] \\
&\quad\Rightarrow\; [(\lambda x.\,\text{alice}\backslash(\text{see}/x))(\underline{\text{who}})] \\
&\quad\Rightarrow\; \underline{[\lambda y.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(y)]]} \\
&\quad\Rightarrow\; \lambda y.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(y)]
\end{aligned}$$

The last line is equivalent to the desired question denotation $\lambda y.\,\text{see}(y)(\text{alice})$, as can be seen by applying it to bob and continuing reducing. Note that *who* is evaluated right after the trace is, and no earlier.

Recall from §5.1 that *everyone's mother* is as genuinely quantificational as *everyone* alone. Similarly, *whose mother* in this treatment of *wh*-questions is as genuinely interrogative as *who* alone. Therefore, raised *wh*-phrases can pied-pipe surrounding material: To generate the sentence

(55) Whose mother did Alice see?

we can place *whose mother* together in a fronted thunk:

(56) $(\lambda().\,\text{who}\backslash\text{mother})\backslash[\text{alice}\backslash(\text{see}/\_\_)]$
$\Rightarrow\ (\lambda().\,\text{who}\backslash\text{mother})\backslash\underline{[\lambda g.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(g())]]}$
$\Rightarrow\ \underline{(\lambda().\,\text{who}\backslash\text{mother})\backslash(\lambda g.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(g())])}$
$\Rightarrow\ [(\lambda x.\,\text{alice}\backslash(\text{see}/x))(\underline{(\lambda().\,\text{who}\backslash\text{mother})()})]$
$\Rightarrow\ [(\lambda x.\,\text{alice}\backslash(\text{see}/x))(\underline{\text{who}}\backslash\text{mother})]$
$\Rightarrow\ \underline{[\lambda y.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(y\backslash\text{mother})]]}$
$\Rightarrow\ \lambda y.\,[(\lambda x.\,\text{alice}\backslash(\text{see}/x))(y\backslash\text{mother})]$

The last line is equivalent to the desired question denotation $\lambda y.\,\text{see}(\text{mother}(y))(\text{alice})$, as can be seen by applying it to bob and continuing reducing. Note that *whose mother* is evaluated right after the trace is, and no earlier.

5.5. **Binding in *wh*-questions.** Another way to make life with the trace more exciting is to put output or input in the fronted phrase. In other words, let us consider when a raised *wh*-phrase can bind a pronoun or contain a pronoun to be bound. For simplicity, I assume that *who* in (57) is a raised *wh*-phrase.

(57) Who$_i$ __ saw his$_i$ mother?

(58) $(\lambda().\,\text{output(who)})\backslash[\_\_\backslash(\text{see}/(\text{input}\backslash\text{mother}))]$
$\Rightarrow\ (\lambda().\,\text{output(who)})\backslash\underline{[\lambda g.\,[(\lambda x.\,x\backslash(\text{see}/(\text{input}\backslash\text{mother})))(g())]]}$
$\Rightarrow\ \underline{(\lambda().\,\text{output(who)})\backslash(\lambda g.\,[(\lambda x.\,x\backslash(\text{see}/(\text{input}\backslash\text{mother})))(g())])}$
$\Rightarrow\ [(\lambda x.\,x\backslash(\text{see}/(\text{input}\backslash\text{mother})))(\underline{(\lambda().\,\text{output(who)})()})]$
$\Rightarrow\ [(\lambda x.\,x\backslash(\text{see}/(\text{input}\backslash\text{mother})))(\text{output}(\underline{\text{who}}))]$
$\Rightarrow\ \underline{[\lambda y.\,[(\lambda x.\,x\backslash(\text{see}/(\text{input}\backslash\text{mother})))(\text{output}(y))]]}$
$\Rightarrow\ \lambda y.\,[(\lambda x.\,x\backslash(\text{see}/(\text{input}\backslash\text{mother})))(\text{output}(y))]$

The last line is equivalent to the desired question denotation $\lambda y.\,\text{see}(\text{mother}(y))(y)$, as can be seen by applying it to bob and continuing reducing. Note that *output(who)* is evaluated when the trace is, which is before *input* is evaluated because the trace occurs before *his*. Binding thus succeeds.

By contrast, binding fails if the trace occurs after the pronoun. This happens in (43b) above and (59) below.

(59) *Who$_i$ did his$_i$ mother see __?

(60) $(\lambda().\,\text{output(who)})\backslash[(\underline{\text{input}}\backslash\text{mother})\backslash(\text{see}/\_\_)]$
$\Rightarrow\ (\lambda().\,\text{output(who)})\backslash\underline{[\lambda x.\,[(x\backslash\text{mother})\backslash(\text{see}/\_\_)]]}$
$\Rightarrow\ (\lambda().\,\text{output(who)})\backslash(\lambda x.\,[(x\backslash\text{mother})\backslash(\text{see}/\_\_)])$

After two reduction steps, we see a type mismatch: the variable x needs to be an individual (of type e) in order to be passed to mother, but $\lambda().\,\text{output(who)}$ is a thunked individual (of type $\langle(), e\rangle$). Binding thus fails as desired, just as in (27).

Furthermore, combining left-to-right evaluation with thunking makes correct predictions not just when the *wh*-phrase itself tries to bind, but also when a sub-phrase of the *wh*-phrase tries to bind.

(61) a. Whose$_i$ friend's$_j$ neighbor$_k$ did Alice think __ saw his$_{i/j/k}$ mother?
b. Whose$_i$ friend's$_j$ neighbor$_k$ did Alice think his$_{*i/*j/*k}$ mother saw __?

Chris and I call these cases "pied-binding".

5.6. **Superiority.** We have just seen how raised *wh*-phrases interact with binding. Raised *wh*-phrases also interact with in-situ *wh*-phrases, under the rubric of superiority. The basic pattern of superiority is shown by the contrast between (62) and (63).

(62) Who __ saw who?

(63) *Who did who see __?

Our theory turns out to already predict these judgments. The difference between these examples is again due to evaluation order.

In the acceptable question (62), the trace is evaluated before the in-situ *who*. The trace makes the gapped clause __ *saw who* call for a thunked individual that is the filler phrase, a need nicely fulfilled by the fronted *who*.

(64)
$$\begin{aligned}
&(\lambda().\,\text{who})\backslash[\underline{\quad}\backslash(\text{see}/\text{who})]\\
&\quad\Rightarrow\ (\lambda().\,\text{who})\backslash\underline{[\lambda g.\,[(\lambda x.\,x\backslash(\text{see}/\text{who}))(g())]]}\\
&\quad\Rightarrow\ \underline{(\lambda().\,\text{who})\backslash(\lambda g.\,[(\lambda x.\,x\backslash(\text{see}/\text{who}))(g())])}\\
&\quad\Rightarrow\ [(\lambda x.\,x\backslash(\text{see}/\text{who}))\underline{((\lambda().\,\text{who})())}]\\
&\quad\Rightarrow\ [(\lambda x.\,x\backslash(\text{see}/\text{who}))(\underline{\text{who}})]\\
&\quad\Rightarrow\ \underline{[\lambda y.\,[(\lambda x.\,x\backslash(\text{see}/\text{who}))(y)]]}\\
&\quad\Rightarrow\ \lambda y.\,[(\lambda x.\,x\backslash(\text{see}/\text{who}))(y)]
\end{aligned}$$

The last line is equivalent to the desired question denotation

(65)
$$\lambda y.\,\lambda z.\,\text{see}(z)(y),$$

as can be seen by applying it to alice and bob and continuing reducing.

In the unacceptable question (63), an in-situ *who* is evaluated first. The in-situ *who* makes the gapped clause *did who see* __ call for an unthunked individual that is the answer to the *wh*-question, which the fronted *who* is not—so evaluation gets stuck, as desired.

(66)
$$\begin{aligned}
&(\lambda().\,\text{who})\backslash[\underline{\text{who}}\backslash(\text{see}/\underline{\quad})]\\
&\quad\Rightarrow\ (\lambda().\,\text{who})\backslash\underline{[\lambda x.\,[x\backslash(\text{see}\underline{\quad})]]}\\
&\quad\Rightarrow\ (\lambda().\,\text{who})\backslash(\lambda x.\,[x\backslash(\text{see}\underline{\quad})])
\end{aligned}$$

One way to understand the problem encountered after two reduction steps above is as a type mismatch: in-situ *who* requires an unthunked individual as input, but what the fronted filler provides is a thunked individual. But even were the raised *who* not thunked, the sentence (63) still does not mean (65). Rather, the expression

(67)
$$\text{who}\backslash[\text{who}\backslash(\text{see}/\underline{\quad})]$$

denotes—reduces to—something incoherent that might be paraphrased "Which x answers the question 'who saw __?'?".

The two examples above show that evaluation order imposes an "intervention effect" (Beck 1996; Pesetsky 2000) or a "minimal link condition" (Chomsky 1995) between the filler and the gap in a multiple-*wh* question.

5.7. **Formalization.** In the presentation so far, I have mainly used example sentences to buttress the intuition that linguistic phenomena can be fruitfully treated as computational side effects. Chris Barker and I have implemented these ideas in a combinatory categorial grammar that fits on a single page (Shan and Barker 2004), and tested them using a parser he wrote. We are unaware of any other implemented parser that deals with quantification, binding, and *wh*-questions at the same time, while ruling out crossover and superiority.

One particularly nice thing about having a machine-executable implementation of our theory is that you can flip a switch—simply enable the right-to-left evaluation rule and disable the left-to-right evaluation rule—and watch the grammar reverse its predictions regarding crossover and superiority. This reversal convinced us that left-to-right evaluation is a viable, unifying explanation for crossover and superiority, and I think an intuitively appealing one as well.

6. Levels of scope-taking

Throughout the discussion above, I have completely neglected the issue of scope ambiguity. As it stands, our grammar generates only one reading for the ambiguous sentence

(68) Someone loves everyone.

The reading that we do predict is surface scope, where *someone* scopes over *everyone*. This prediction is because we stipulate left-to-right evaluation, and quantifiers evaluated earlier scope wider.

(69)
$$\begin{aligned}
&(\underline{\xi f.\, \exists x.\, f(x)})\backslash(\mathrm{love}/(\xi f.\, \forall y.\, f(y))) \\
&\quad\Rightarrow\ \exists x.\, \underline{(\lambda z.\, [z\backslash(\mathrm{love}/(\xi f.\, \forall y.\, f(y)))])(x)} \\
&\quad\Rightarrow\ \exists x.\, [x\backslash(\mathrm{love}/(\underline{\xi f.\, \forall y.\, f(y)}))] \\
&\quad\Rightarrow\ \exists x.\, [\forall y.\, \underline{(\lambda z.\, [x\backslash(\mathrm{love}/z)])(y)}] \\
&\quad\Rightarrow\ \exists x.\, [\forall y.\, [x\underline{\backslash}(\mathrm{love}\underline{/}y)]] \\
&\quad\Rightarrow\ \exists x.\, [\forall y.\, \underline{[\mathrm{love}(y)(x)]}] \\
&\quad\Rightarrow\ \exists x.\, \underline{[\forall y.\, \mathrm{love}(y)(x)]} \\
&\quad\Rightarrow\ \exists x.\, \forall y.\, \mathrm{love}(y)(x)
\end{aligned}$$

The shift for *someone* is evaluated before the shift for *everyone*, so the former quantifier dictates the outermost shape of the final result. No matter what evaluation order we specify, as long as our semantic rules remain deterministic, they will only generate one reading for the sentence, never both.

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 route has been pursued with some success by Barker (2002) and de Groote (2001), but it contradicts the unified account of crossover and superiority in §5 above. A second way to introduce nondeterminism is to maintain the hypothesis that natural language expressions are evaluated from left to right, but allow multiple *context levels* (Barker 2000; Danvy and Filinski 1990) to keep multiple side effects out of each other's way. I now explain this second way.

As introduced in §2, shift in our metalanguage only affects up to the closest enclosing reset. Multiple context levels relax this restriction by placing a subscript on each shift and reset operator. The effect of an nth-level shift, written $\xi_n f.\, e$ where f is a variable name and e is an expression, is restricted to the closest enclosing mth-level reset, written $[_m \cdots]$, such that $m \leq n$. The following reductions illustrate.

(70)
$$\begin{aligned}
&\text{“real ”} + [_1 \text{“compositional ”} + (\xi_1 f.\, \text{“directly ”} + f(\text{“semantics”}))] \\
&\quad\Rightarrow\ \cdots\ \Rightarrow\ \text{“real directly compositional semantics”}
\end{aligned}$$

(71)
$$\begin{aligned}
&\text{“real ”} + [_1 \text{“compositional ”} + (\xi_0 f.\, \text{“directly ”} + f(\text{“semantics”}))] \\
&\quad\Rightarrow\ \cdots\ \Rightarrow\ \text{“directly real compositional semantics”}
\end{aligned}$$

(72) “real ” + [$_0$“compositional ” + ($\xi_1 f$. “directly ” + f(“semantics”))]
$\Rightarrow \cdots \Rightarrow$ “real directly compositional semantics”

(73) “real ” + [$_0$“compositional ” + ($\xi_0 f$. “directly ” + f(“semantics”))]
$\Rightarrow \cdots \Rightarrow$ “real directly compositional semantics”

Danvy and Filinski (1990) give a denotational semantics for shift and reset at multiple context levels, using multiple levels of continuations. Taking advantage of their work, we let quantifiers manipulate the context at any level, say the nth level.

(74) $\text{everyone}_n = \xi_n f.\, \forall x.\, f(x)$

(75) $\text{someone}_n = \xi_n f.\, \exists x.\, f(x)$

In other words, we posit that each occurrence of *everyone* and *someone* is ambiguous as to the level n at which it shifts.

The ambiguity of (68) is now predicted as follows. Suppose that *someone* shifts at the mth level and *everyone* shifts at the nth level.

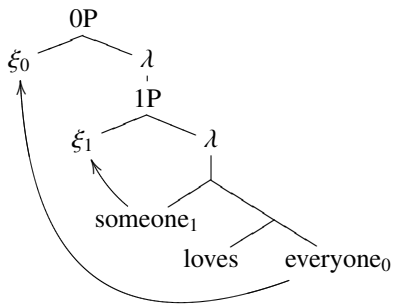
(76) Someone_m loves everyone_n.

If $m \leq n$, the surface scope reading results. If $m > n$, the inverse scope reading results. For example, when $m = 1$ and $n = 0$, the inverse scope is computed by the following reductions.

(77)
$$\begin{aligned}
&\underline{\text{someone}_1}\backslash(\text{love}/\text{everyone}_0)\\
&\Rightarrow \exists x.\, \underline{(\lambda z.\, [_1 z\backslash(\text{love}/\text{everyone}_0)])(x)}\\
&\Rightarrow \exists x.\, [_1 x\backslash(\text{love}/\underline{\text{everyone}_0})]\\
&\Rightarrow \forall y.\, \underline{(\lambda z.\, [_0 \exists x.\, [_1 x\backslash(\text{love}/z)]])(y)}\\
&\Rightarrow \forall y.\, [_0 \exists x.\, [_1 x\underline{\backslash}(\text{love}\underline{/}y)]]\\
&\Rightarrow \forall y.\, [_0 \exists x.\, \underline{[_1 \text{love}(y)(x)]}]\\
&\Rightarrow \forall y.\, \underline{[_0 \exists x.\, \text{love}(y)(x)]}\\
&\Rightarrow \forall y.\, \exists x.\, \text{love}(y)(x)
\end{aligned}$$

This analysis of inverse scope may be reminiscent of accounts that posit a hierarchy of functional projections at the clausal boundary. We can think of each context level as a functional projection, which attracts quantifiers destined for that level or any inner level.

(78)



The same hierarchy of context levels allows in-situ *wh*-phrases to take scope ambiguously (Baker 1968).

(79) Who remembers where we bought what?
a. Alice remembers where we bought the vase.
b. Alice remembers where we bought what.

7. Conclusion

To sum up, let me use the linguistic analyses in this paper as a poster child for the broader agenda that I set out at the beginning—that is, to relate computational and linguistic side effects. I certainly don't think that Chris Barker and I have now completely uncovered the mysteries of crossover and superiority, but to the extent that our approach is successful, it epitomizes three goals on my agenda, namely *uniformity*, *interaction*, and *evaluation*.

7.1. **Uniformity.** All linguistic side effects can be treated under the same framework. In this paper, that framework is the metalanguage of shift and reset. When we added to our grammar lexical items for quantification, binding, *wh*-questions, and so on, all we did was that—adding lexical items. We did not have to revise our composition rules or re-lift our semantic types. From the analogy between linguistic and computational side effects, we can derive intuition for how to treat all of the settings in natural language where referential transparency seems to be at stake.

Of course, even though I just used the word "can", there is always the possibility that the pieces may not all fall into place by themselves as we work on our grand uniform theory of linguistic side effects. They probably won't—at least not so easily. For now, I can only make a methodological observation: isn't it curious how we developed this nice semantic theory by working with not just the pure λ-calculus but also shift and reset? You can also further speculate that this curiosity tells us about how human language works.

Some readers may complain that shift and reset can treat so many linguistic side effects only because they are so powerful—too powerful to yield any insight or constraint on natural language. But as I emphasized in §4, shift and reset are just metalinguistic shorthand for continuations, which are a generalization of good old generalized quantifiers. Shift and reset, and their link to computational side effects, tell us how to fully exploit the power that already comes with generalized quantifiers, without further complicating the basic machinery of syntax and semantics with possible worlds for intensionality, variables for binding, alternatives for focus, undefinedness for presupposition, and so on.

7.2. **Interaction.** Because all our analyses are phrased in terms of the same framework, there is at least a shred of hope that, if we just bang some lexical items that deal with quantification against some other items that deal with binding, we would get an harmonious analysis of quantification and binding. In the cases surveyed in this paper, this shred of hope did work out, though there were technical details that needed to be dealt with, which for me was made easier by the connection to computational side effects.

By the way, the picture is not entirely rosy on the programming-language side either. The engineering ideal would be for the issues of multiple, interacting computational side effects to be solved to the extent where you can just throw together any number of arbitrary side effects and get a working programming language by stirring the mix. That would make it much easier for people to design new programming languages as well as understand existing ones, but there are unresolved technical difficulties. Who knows—perhaps these technical difficulties in programming language engineering correspond to empirical generalizations or constraints in natural languages, however skilled speakers of natural languages and users of programming language may be at getting around them.

7.3. **Evaluation.** One thing that I found helpful as I studied programming language semantics is how it relates denotational and operational models of meaning, as I mentioned earlier in §4. When a computer scientists says that some program expression denotes some

value, there is no philosophical commitment being made as to what the "true meanings" of programs are. A lot of insight and understanding can often be gained by spelling out *two* semantics for the same language and proving them equivalent. For example, one semantics might be denotational in that it assigns a semantic value to each expression, and another might be operational in that it specifies reductions on terms that correspond to how the program might be executed by a computer. These are both valid viewpoints.

In the linguistics I just presented, there are three places where a link between denotational and operational semantics helps us formulate a theory that gives insight and makes correct predictions.

(1) The shift-reset metalanguage. It is often easier for me to understand what's going on by looking at a shift-reset expression instead of a gazillion λs in the pure λ-calculus.
(2) Left-to-right evaluation. Once we take a naïve processing explanation and couch it in terms of denotations (with the help of continuation semantics), we can explain cases (such as those in §5.5) where the empirical data is the exact opposite of what would be predicted by the naïve left-to-right account.
(3) Thunking. We used thunking to analyze overt *wh*-fronting in §5.4: it corresponds to delayed evaluation, which postpones the side effects incurred by a program expression.

Of course, the insight that linguistic expressions are like program instructions is already present in dynamic semantics (Groenendijk and Stokhof 1991; Heim 1982; Kamp 1981; see also Moschovakis 1993). Dynamic semantics has been applied to a variety of natural-language phenomena, such as basic crossover and verb-phrase ellipsis (van Eijck and Francez 1995; Gardent 1991; Hardt 1999). However, dynamic semantics alone does not account for interactions among binding, quantification, and *wh*-questions. The work described here enables dynamic semantics to compose meanings at the intrasentential level, and generalizes it to side effects other than binding, existential quantification, and presupposition.

It might be strange for me to speak of semantics that are not denotational, especially since the topic of this workshop, "direct compositionality", pretty much presupposes that a semantic theory is one that assigns denotations to phrases. But denotations don't have to be sets, and composition doesn't have to be function application. For example, I find it promising that *game semantics* is taking hold in both programming language semantics and natural language semantics (Abramsky and McCusker 1997; inter alia). In linguistics, a lot of questions are being asked about the proper line, if any, to draw between semantics and pragmatics, or between semantics and syntax. In computer science, as it turned out, even something as innocent-looking as the pure λ-calculus can be decomposed and analyzed fruitfully as a model of computation by interaction. These are bona fide semantics, even though they are not model-theoretic in the original Montagovian sense, but closer to a proof-theoretic ideal.

7.4. **Farewell.** So, there you have it: uniformity, interaction, and evaluation. These are what I think we can achieve and clarify in our theory of linguistic side effects in natural languages, by drawing an analogy with computational side effects in programming languages. I hope that I have given you some concrete examples of this approach, in the form of proposed theories that are, I think, improvements over their predecessors (and I mean "predecessors" in terms of both time and intellectual heritage).

References

Abramsky, Samson, and Guy McCusker. 1997. Game semantics. Presented at the 1997 Marktoberdorf Summer School. 26 Sep. 2000, http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/mdorf97.ps.gz.

Baker, Carl Leroy. 1968. Indirect questions in English. Ph.D. thesis, University of Illinois.

Barker, Chris. 2000. Notes on higher-order continuations. Manuscript, University of California, San Diego.

———. 2002. Continuations and the nature of quantification. *Natural Language Semantics* 10(3):211–242.

Beck, Sigrid. 1996. Quantified structures as barriers for LF movement. *Natural Language Semantics* 4:1–56.

den Besten, Hans, and Gert Webelhuth. 1990. Stranding. In *Scrambling and barriers*, ed. Günther Grewendorf and Wolfgang Sternefeld, 77–92. Amsterdam: John Benjamins.

Büring, Daniel. 2001. A situation semantics for binding out of DP. In *Proceedings from Semantics and Linguistic Theory XI*, ed. Rachel Hastings, Brendan Jackson, and Zsofia Zvolensky, 56–75. Ithaca: Cornell University Press.

Chomsky, Noam. 1995. *The minimalist program*, chap. 4 (Categories and transformations), 219–394. Cambridge: MIT Press.

Cinque, Guglielmo. 1990. *Types of $\bar{A}$-dependencies*. Cambridge: MIT Press.

Cooper, Robin. 1983. *Quantification and syntactic theory*. Dordrecht: Reidel.

Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark. http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz.

———. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.

———. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

van Eijck, Jan, and Nissim Francez. 1995. Verb-phrase ellipsis in dynamic semantics. In *Applied logic: How, what, and why: Logical approaches to natural language*, ed. László Pólos and Michael Masuch. Dordrecht: Kluwer.

Felleisen, Matthias. 1987. The calculi of λ_v-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Computer Science Department, Indiana University. Also as Tech. Rep. 226.

———. 1988. The theory and practice of first-class prompts. In *POPL '88: Conference record of the annual ACM symposium on principles of programming languages*, 180–190. New York: ACM Press.

Filinski, Andrzej. 1994. Representing monads. In *POPL '94: Conference record of the annual ACM symposium on principles of programming languages*, 446–457. New York: ACM Press.

———. 1996. Controlling effects. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-96-119.

———. 1999. Representing layered monads. In *POPL '99: Conference record of the annual ACM symposium on principles of programming languages*, 175–188. New York: ACM Press.

Frege, Gottlob. 1891. Funktion und begriff. Vortrag, gehalten in der Sitzung vom 9. Januar 1891 der Jenaischen Gesellschaft für Medizin und Naturwissenschaft, Jena: Hermann Pohle. English translation Frege (1980a).

———. 1892. Über sinn und bedeutung. *Zeitschrift für Philosophie und philosophische Kritik* 100:25–50. English translation Frege (1980b).

———. 1980a. Function and concept. In Frege (1980c), 21–41. Also in Frege (1997), 130–148.

———. 1980b. On sense and reference. In Frege (1980c), 56–78. Also in Frege (1997), 151–171.

———. 1980c. *Translations from the philosophical writings of Gottlob Frege*, ed. Peter Geach and Max Black. 3rd ed. Oxford: Blackwell.

———. 1997. *The Frege reader*, ed. Michael Beaney. Oxford: Blackwell.

Gardent, Claire. 1991. Dynamic semantics and VP-ellipsis. In *Logics in AI: European workshop JELIA '90*, ed. Jan van Eijck, 251–266. Lecture Notes in Artificial Intelligence 478, Berlin: Springer-Verlag.

Groenendijk, Jeroen, and Martin Stokhof. 1991. Dynamic predicate logic. *Linguistics and Philosophy* 14(1):39–100.

de Groote, Philippe. 2001. Type raising, continuations, and classical logic. In *Proceedings of the 13th Amsterdam Colloquium*, ed. Robert van Rooy and Martin Stokhof, 97–101. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Hamblin, Charles Leonard. 1973. Questions in Montague English. *Foundations of Language* 10:41–53.

Hardt, Daniel. 1999. Dynamic interpretation of verb phrase ellipsis. *Linguistics and Philosophy* 22(2):185–219.

Hatcliff, John, and Olivier Danvy. 1997. Thunks and the λ-calculus. *Journal of Functional Programming* 7(3):303–319.

Heim, Irene. 1982. The semantics of definite and indefinite noun phrases. Ph.D. thesis, Department of Linguistics, University of Massachusetts.

Ingerman, Peter Zilahy. 1961. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM* 4(1):55–58.

Jacobson, Pauline. 1999. Towards a variable-free semantics. *Linguistics and Philosophy* 22(2):117–184.

———. 2000. Paycheck pronouns, Bach-Peters sentences, and variable-free semantics. *Natural Language Semantics* 8(2):77–155.

Kameyama, Yukiyoshi, and Masahito Hasegawa. 2003. A sound and complete axiomatization of delimited continuations. In *ICFP '03: Proceedings of the ACM international conference on functional programming*. New York: ACM Press.

Kamp, Hans. 1981. A theory of truth and semantic representation. In *Formal methods in the study of language: Proceedings of the 3rd Amsterdam Colloquium*, ed. Jeroen A. G. Groenendijk, Theo M. V. Janssen, and Martin B. J. Stokhof, 277–322. Amsterdam: Mathematisch Centrum.

Keller, William R. 1988. Nested Cooper storage: The proper treatment of quantification in ordinary noun phrases. In *Natural language parsing and linguistic theories*, ed. Uwe Reyle and Christian Rohrer, 432–447. Dordrecht: Reidel.

May, Robert C. 1977. The grammar of quantification. Ph.D. thesis, Department of Linguistics and Philosophy, Massachusetts Institute of Technology. Reprinted by New York: Garland, 1991.

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.

Moortgat, Michael. 1988. *Categorial investigations: Logical and linguistic aspects of the Lambek calculus*. Dordrecht: Foris.

———. 1995. In situ binding: A modal analysis. In *Proceedings of the 10th Amsterdam Colloquium*, ed. Paul Dekker and Martin Stokhof, 539–549. Institute for Logic, Language and Computation, Universiteit van Amsterdam.

———. 1996. Generalized quantification and discontinuous type constructors. In *Discontinuous constituency*, ed. Harry C. Bunt and Arthur van Horck, 181–207. Berlin: Mouton de Gruyter.

Moschovakis, Yiannis N. 1993. Sense and denotation as algorithm and value. In *Logic Colloquium '90: Association of Symbolic Logic summer meeting*, ed. Juha Oikkonen and Jouko Vaananen, 210–249. Lecture Notes in Logic 2, Poughkeepsie, NY: Association for Symbolic Logic. Reprinted by Natick, MA: A K Peters.

Pesetsky, David. 2000. *Phrasal movement and its kin*. Cambridge: MIT Press.

Postal, Paul Martin. 1998. *Three investigations of extraction*. Cambridge: MIT Press.

Quine, Willard Van Orman. 1960. *Word and object*. Cambridge: MIT Press.

Raymond, Eric S., et al., eds. 2003. *The jargon file*. `http://www.catb.org/~esr/jargon/`. Version 4.4.7. Version 4.0.0 was published by MIT Press as *The New Hacker's Dictionary*.

Richards, Norvin W., III. 1997. What moves where when in which language? Ph.D. thesis, Department of Linguistics and Philosophy, Massachusetts Institute of Technology. Published as *Movement in Language: Interactions and Architectures* by Oxford University Press, 2001.

Shan, Chung-chieh. 2002. A continuation semantics of interrogatives that accounts for Baker's ambiguity. In *Proceedings from Semantics and Linguistic Theory XII*, ed. Brendan Jackson, 246–265. Ithaca: Cornell University Press.

———. 2004. Shift to control. In *Proceedings of the 5th workshop on Scheme and functional programming*, ed. Olin Shivers and Oscar Waddell, 99–107. Tech. Rep. 600, Computer Science Department, Indiana University.

Shan, Chung-chieh, and Chris Barker. 2002. A unified explanation for crossover and superiority in a theory of binding by predicate abstraction. Poster presented at the North East Linguistic Society conference (NELS 33).

———. 2004. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy*. To appear.

Søndergaard, Harald, and Peter Sestoft. 1990. Referential transparency, definiteness and unfoldability. *Acta Informatica* 27:505–517.

———. 1992. Non-determinism in functional languages. *The Computer Journal* 35(5): 514–523.

Division of Engineering and Applied Sciences, Harvard University
E-mail address: `ccshan@post.harvard.edu`
URL: `http://www.eecs.harvard.edu/~ccshan/`