Inverse scope as metalinguistic quotation in operational semantics

Chung-chieh Shan

Rutgers University
ccshan@rutgers.edu

Abstract. We model semantic interpretation *operationally*: constituents interact as their combination in discourse evolves from state to state. The states are recursive data structures and evolve from step to step by context-sensitive rewriting. These notions of *context* and *order* let us explain inverse-scope quantifiers and their polarity sensitivity as metalinguistic quotation of the wider scope.

1 Introduction

An utterance is both an action and a product [24]. On one hand, when we use an utterance, it acts on the world: its parts affect us and our conversation. For example, a referring expression reminds us of its referent and adjusts the discourse context so that a pronoun later may refer to it. On the other hand, linguistics describes the syntax and semantics of a sentence as a mathematical object: a product, perhaps a tree or a function. But trees and functions do not remind or adjust; at their most active, they merely contain or map. How then does the meaning of an utterance determine its effects on the discourse and its participants? In particular, how do utterances affect their *context*, and in the *order* that they do?

We approach this "mind-body problem" the way many programming-language semanticists approach an analogous issue. On one hand, a program such as

$$2 \to n;\ n \cdot 3 \tag{1}$$

expresses a sequence of actions: store the number 2 in the memory location n, then retrieve the contents of n and multiply it by 3. On the other hand, computer science describes the syntax and semantics of the program as a tree or a function. To view the same program as both a dynamic action and a static product, programming-language theory uses *operational semantics* alongside denotational semantics. A denotational semantics assigns a denotation to each expression—compositionally, by specifying the denotation of a complex expression in terms of the denotations of its parts. In contrast, an operational semantics defines a *transition relation* on states—for each state, what it can become after one step of computation [16]. Technical conditions relate denotational and operational semantics. For example, *adequacy* requires that two utterances with the same denotation must have the same operational outcome.

To analyze the example (1), we could define a state as an ordered pair of an integer n and a current program, and the transition relation $\rightsquigarrow$ as a set that includes the three transitions

$$\langle 0,\ (2 \to n;\ n \cdot 3)\rangle \rightsquigarrow \langle 2,\ n \cdot 3\rangle \rightsquigarrow \langle 2,\ 2 \cdot 3\rangle \rightsquigarrow \langle 2,\ 6\rangle. \tag{2}$$

These transitions together say that the program (1) starting with the memory location n initialized to 0 yields the final result 6 with the contents of n changed to 2. Unlike in denotational semantics, we specify the outcome of a program without recursively specifying what each part of the program denotes. Rather, we specify rewriting rules such as

$$\langle x,\ (y \to n;\ P)\rangle \rightsquigarrow \langle y,\ P\rangle \quad \text{for any numbers } x, y \text{ and any program } P \tag{3}$$

to be elements of the transition relation, so that the first computation step in (2) goes through. In other words, we include

$$\left\{ \left\langle \langle x,\ (y \to n;\ P)\rangle,\ \langle y,\ P\rangle \right\rangle \mid x \text{ and } y \text{ are two numbers; } P \text{ is a program} \right\} \tag{4}$$

in the transition relation as a subset.

Following a longstanding analogy between natural and formal languages, we model natural-language semantics operationally. In Section 2, we review a standard operational semantics for a λ-calculus with *delimited control*. In Section 3, we use this operational semantics to explain quantification and polarity sensitivity. We focus on these applications because they go beyond phenomena such as anaphora and presupposition, where dynamic semantics has long been fruitful. In particular, metalinguistic quotation (or *multistage programming*) extends our account to inverse scope. We conclude in Section 4.

2 Context and order

We use the λ-calculus to introduce the notions of *context* and *order*, then let an expression interact actively with its context. These computational notions recur in various linguistic guises, especially in analyses of quantification, as Section 3 makes concrete.

2.1 Expressions and contexts

The set of λ-calculus expressions is recursively defined by the following grammar. In words, an expression is either a variable x, an abstraction $\lambda x.\, E$, or an application EE.

$$E \to x \qquad E \to \lambda x.\, E \qquad E \to EE \tag{5}$$

The derivation below shows that $\lambda x.\, xx$ is an expression, even though no function x can apply to itself in standard set theory.

$$E \to \lambda x.\, E \to \lambda x.\, EE \to \lambda x.\, Ex \to \lambda x.\, xx \tag{6}$$

In programming practice, constants such as 2 and multiplication $\cdot$ are also expressions. Not shown in the grammar is the convention that we equate expressions that differ only by variable renaming, so $\lambda x.\, \lambda x.\, x = \lambda x.\, \lambda y.\, y = \lambda y.\, \lambda x.\, x \neq \lambda x.\, \lambda y.\, x$.

We specify certain expressions to be *values*, namely those generated by the following grammar. In words, a value is any variable or abstraction, but not an application.

$$V \to x \qquad\qquad V \to \lambda x.\, E \tag{7}$$

For example, the identity expression $\lambda x.\, x$ (henceforth I) is a value. In practice, constants such as 2 and multiplication $\cdot$ are also values. Intuitively, a value is an expression that is done computing, so the expression $2 \cdot 3$ is not a value.

An operational semantics is a relation on states. (More precisely, we consider *small-step* operational semantics.) Our states are just closed expressions (that is, expressions with no unbound variables), and the transition relation is in fact a partial function that maps each closed expression to what it becomes after one step of computation. We define the transition relation as follows. A *context* is an expression with a gap, which we write as $[\,]$. For example, $\lambda x.\, x(y[\,])$ is a context. The set of *evaluation contexts* $C[\,]$ is defined by the grammar below, in the style of Felleisen [5]. The notation $C[\ldots]$ means to replace the gap $[\,]$ in the context $C[\,]$ by an expression or another context.

$$C[\,] \to [\,] \qquad C[\,] \to C\big[[\,]E\big] \qquad C[\,] \to C\big[V[\,]\big] \tag{8}$$

The first production above says that the null context $[\,]$—the context with nothing but a gap—is an evaluation context. The second production says that, whenever $C[\,]$ is an evaluation context, replacing its gap by $[\,]E$ (that is, the application of a gap to any expression E) gives another evaluation context. The third production says that, whenever $C[\,]$ is an evaluation context, replacing its gap by $V[\,]$ (that is, the application of any value V to a gap) gives another evaluation context. For example, the derivation below shows that $I\big([\,](II)\big)$ is an evaluation context. (Recall from above that I stands for $\lambda x.\, x$.)

$$C[\,] \to C\big[[\,](II)\big] \to C\big[I\big([\,](II)\big)\big] \to I\big([\,](II)\big) \tag{9}$$

We now define the transition relation $\rightsquigarrow$ to be the set of expression-pairs

$$\{\, \langle C[(\lambda x.\, E)V],\, C[E']\rangle \mid C[\,] \text{ is an evaluation context; } E' \text{ substitutes the value } V \text{ for the variable } x \text{ in the expression } E \,\}, \tag{10}$$

or for short,

$$C[(\lambda x.\, E)V] \rightsquigarrow C[E\,\{x \mapsto V\}]. \tag{11}$$

In words, a transition is a λ-conversion in an evaluation context where the argument is a value expression. Letting $C[\,] = I\big([\,](II)\big)$, $V = I$, and $E = x$ gives

$$I\big((II)(II)\big) \rightsquigarrow I\big(I(II)\big). \tag{12}$$

In fact, this is the only step that a computation starting at $I\big((II)(II)\big)$ can take. It takes three more steps to reach a value, namely I:

$$I\big(I(II)\big) \rightsquigarrow I(II) \rightsquigarrow II \rightsquigarrow I. \tag{13}$$

In practice, λ-conversions are joined by other transitions such as $1+2\cdot 3 \rightsquigarrow 1+6$.

This operational semantics is not the only reasonable one for the λ-calculus. Instead of or in addition to the transition (12), $I\big((II)(II)\big)$ could transition to $(II)(II)$ or $I\big((II)I\big)$. Different operational semantics regulate the *order* in which to run parts of a program differently, by constraining the set of evaluation contexts and the rewriting that takes place inside. Our transition relation in (10)–(11) implements a *call-by-value*, *left-to-right* evaluation order: it only performs λ-conversion when the argument is a value (the V in (10)–(11) rules out a transition to $(II)(II)$) and everything to the left is also a value (the V in (8) rules out a transition to $I\big((II)I\big)$).

2.2 Delimited control

The simple expression $I\big((II)(II)\big)$ above is not sensitive to the evaluation order: pretty much any reasonable order will bring it to the final value I after a few transitions. However, as we add functionality to the λ-calculus as a programming language, different operational semantics result in different program outcomes. A particularly powerful and linguistically relevant addition is *delimited control* [5, 6], which lets an expression manipulate its context. To illustrate, we add Danvy and Filinski's delimited-control operators *shift* and *reset* [2, 3, 4] to the λ-calculus.

Before specifying shift and reset formally, let us first examine some example transition sequences among arithmetic expressions. Reset by itself does not perform any computation, so the programs $1 + 2 \cdot 3$ and $1 + \mathrm{reset}(2 \cdot 3)$ yield the same final value:

$$1 + 2 \cdot 3 \rightsquigarrow 1 + 6 \rightsquigarrow 7, \tag{14}$$

$$1 + \mathrm{reset}(2 \cdot 3) \rightsquigarrow 1 + \mathrm{reset}\,6 \rightsquigarrow 1 + 6 \rightsquigarrow 7. \tag{15}$$

Shift means to *remove* the surrounding context—up to the nearest enclosing reset—into a variable. This functionality lets an expression manipulate its context. For example, the variable f below receives the context that multiplies by 2, so the program computes $1 + 3 \cdot 2 \cdot 2 \cdot 5$.

$$\begin{aligned}
&1 + \mathrm{reset}\big(2 \cdot \mathrm{shift}\, f.\, (3 \cdot f(f(5)))\big) \\
&\quad\rightsquigarrow 1 + \mathrm{reset}\big(3 \cdot (\lambda x.\, \mathrm{reset}(2 \cdot x))((\lambda x.\, \mathrm{reset}(2 \cdot x))(5))\big) \\
&\quad\rightsquigarrow 1 + \mathrm{reset}\big(3 \cdot (\lambda x.\, \mathrm{reset}(2 \cdot x))(\mathrm{reset}(2 \cdot 5))\big) \\
&\quad\rightsquigarrow 1 + \mathrm{reset}\big(3 \cdot (\lambda x.\, \mathrm{reset}(2 \cdot x))(\mathrm{reset}\,10)\big) \\
&\quad\rightsquigarrow 1 + \mathrm{reset}\big(3 \cdot (\lambda x.\, \mathrm{reset}(2 \cdot x))10\big) \\
&\quad\rightsquigarrow 1 + \mathrm{reset}\big(3 \cdot \mathrm{reset}(2 \cdot 10)\big) \rightsquigarrow 1 + \mathrm{reset}(3 \cdot \mathrm{reset}\,20) \\
&\quad\rightsquigarrow 1 + \mathrm{reset}(3 \cdot 20) \rightsquigarrow 1 + \mathrm{reset}\,60 \rightsquigarrow 1 + 60 \rightsquigarrow 61
\end{aligned} \tag{16}$$

To take another example, the shift expression below does not use the variable f and so discards its surrounding context and supplies the reset with the result 4 right away.

$$1 + \mathrm{reset}\big(2 \cdot 3 \cdot (\mathrm{shift}\, f.\, 4) \cdot 5\big) \rightsquigarrow 1 + \mathrm{reset}\, 4 \rightsquigarrow 1 + 4 \rightsquigarrow 5 \tag{17}$$

We call reset a *control delimiter* because it delimits how much context an enclosed shift expression manipulates.

For concreteness, the rest of this subsection formalizes the delimited-control operators shift and reset; it can be skipped if the examples above suffice. We add two productions for expressions.

$$E \to \mathrm{reset}\, E \qquad\qquad E \to \mathrm{shift}\, f.\, E \tag{18}$$

The expression "shift $f.\, E$" binds the variable f in the body E, so for instance the expressions "shift $f.\, f$" and "shift $x.\, x$" are equal because they differ only by variable renaming.

We add one production for evaluation contexts.

$$C[\,] \to C\big[\mathrm{reset}[\,]\big] \tag{19}$$

We call $D[\,]$ a *subcontext* if it is an evaluation context built without this new production. Finally, we add two new kinds of transitions to our transition relation, one for reset and one for shift:

$$\{\langle C\big[\mathrm{reset}\, V\big],\, C\big[V\big]\rangle \mid C[\,] \text{ is an evaluation context and } V \text{ is a value}\,\}, \tag{20}$$

$$\begin{aligned}\{\langle C\big[\mathrm{reset}\, D[\mathrm{shift}\, f.\, E]\big],\, C\big[\mathrm{reset}\, E'\big]\rangle \\ \mid C[\,] \text{ is an evaluation context; } D[\,] \text{ is a subcontext;} \\ E' \text{ substitutes } \lambda x.\, \mathrm{reset}\, D[x] \text{ for the variable } f \\ \text{in the expression } E, \text{ where } x \text{ is a fresh variable}\,\},\end{aligned} \tag{21}$$

or for short,

$$C\big[\mathrm{reset}\, V\big] \rightsquigarrow C\big[V\big], \tag{22}$$

$$C\big[\mathrm{reset}\, D[\mathrm{shift}\, f.\, E]\big] \rightsquigarrow C\big[\mathrm{reset}\, E\,\{f \mapsto \lambda x.\, \mathrm{reset}\, D[x]\}\big]. \tag{23}$$

3 Linguistic applications

Linguistic theory traditionally views syntax, semantics, and pragmatics as a pipeline, in which semantics maps expressions to denotations. We envision a semantic theory for natural language that is operational in the sense that it specifies transitions among representations rather than mappings to denotations. That is, whereas denotational semantics interprets a syntactic constituent (such as a verb phrase) by mapping it to a separate semantic domain (such as of functions), operational semantics interprets a constituent by rewriting it to other constituents. The rewriting makes sense insofar as the constituents represent real objects. Thus we may specify a fragment of operational semantics as a set of states, a transition relation over the states, and an ideally trivial translation from utterances to states.

3.1 Quantification

We now use delimited control to model quantification. We assume that the sentence

(24) Somebody saw everybody

translates to the program (state)

$$\text{reset}\big(\overbrace{(\text{shift}\, f.\, \exists x.\, fx)}^{\textit{somebody}}\backslash\overbrace{(\text{saw}}^{\textit{saw}}\,/\overbrace{(\text{shift}\, g.\, \forall y.\, gy)}^{\textit{everybody}})\big), \tag{25}$$

where $\backslash$ and $/$ indicate backward and forward function application. This program then makes the following transitions to yield the surface-scope reading of (25).

$$\begin{aligned}
&\rightsquigarrow \text{reset}\big(\exists x.\, (\lambda x.\, \text{reset}(x\backslash(\text{saw}/\,\text{shift}\, g.\, \forall y.\, gy)))x\big)\\
&\rightsquigarrow \text{reset}\big(\exists x.\, \text{reset}(x\backslash(\text{saw}/\,\text{shift}\, g.\, \forall y.\, gy))\big)\\
&\rightsquigarrow \text{reset}\big(\exists x.\, \text{reset}(\forall y.\, (\lambda y.\, \text{reset}(x\backslash(\text{saw}/y)))y)\big)\\
&\rightsquigarrow \text{reset}\big(\exists x.\, \text{reset}(\forall y.\, \text{reset}(x\backslash(\text{saw}/y)))\big)\\
&\rightsquigarrow \text{reset}\big(\exists x.\, \text{reset}(\forall y.\, x\backslash(\text{saw}/y))\big)\\
&\rightsquigarrow \text{reset}\big(\exists x.\, \forall y.\, x\backslash(\text{saw}/y)\big) \rightsquigarrow \exists x.\, \forall y.\, x\backslash(\text{saw}/y).
\end{aligned}$$

The last three transitions assume that the final formula and its subformulas are values.

This example illustrates that the context of a shift operator is the scope of a quantifier, and the order in which expressions are evaluated is that in which they take scope. This analogy is appealing because it extends to other apparently noncompositional phenomena [1, 20–22] and tantalizing because it links meaning to processing.

3.2 Polarity sensitivity

We now extend the model above to capture basic polarity sensitivity: a polarity item such as *anybody* needs to take scope (right) under a polarity licensor such as existential *nobody*.

(26) Nobody saw anybody.

(27) Nobody saw everybody.

(28) *Somebody saw anybody.

Following Fry [8, 9], we treat a polarity license ℓ as a dynamic resource [14] that is produced by *nobody*, required by *anybody*, and disposed of implicitly. To this end, we translate *nobody* to

$$\text{shift}\, f.\, \neg\exists x.\, fx\ell, \tag{29}$$

translate *anybody* to

$$\text{shift}\, g.\, \lambda\ell.\, \exists y.\, gy\ell, \tag{30}$$

and add license-disposal transitions of the form

$$C[V\ell] \rightsquigarrow C[V]. \tag{31}$$

In (30), $\lambda\ell$ denotes a function that must take the constant ℓ as argument.

The sentence (26) translates and computes as follows. The penultimate transition disposes of the used license using (31).

$$\begin{aligned}
(32)\qquad & \text{reset}\big(\overbrace{(\text{shift}\, f.\, \neg\exists x.\, fx\ell)}^{nobody}\backslash\overbrace{(\text{saw}}^{saw}/\overbrace{(\text{shift}\, g.\, \lambda\ell.\, \exists y.\, gy\ell)}^{anybody})\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, (\lambda x.\, \text{reset}(x\backslash(\text{saw}/\,\text{shift}\, g.\, \lambda\ell.\, \forall y.\, gy\ell)))x\ell\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, \text{reset}(x\backslash(\text{saw}/\,\text{shift}\, g.\, \lambda\ell.\, \forall y.\, gy\ell))\ell\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, \text{reset}(\lambda\ell.\, \forall y.\, (\lambda y.\, \text{reset}(x\backslash(\text{saw}/y)))y\ell)\ell\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, (\lambda\ell.\, \forall y.\, (\lambda y.\, \text{reset}(x\backslash(\text{saw}/y)))y\ell)\ell\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, \forall y.\, (\lambda y.\, \text{reset}(x\backslash(\text{saw}/y)))y\ell\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, \forall y.\, \text{reset}(x\backslash(\text{saw}/y))\ell\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, \forall y.\, x\backslash(\text{saw}/y)\ell\big)\\
&\quad\rightsquigarrow \text{reset}\big(\neg\exists x.\, \forall y.\, x\backslash(\text{saw}/y)\big) \rightsquigarrow \neg\exists x.\, \forall y.\, x\backslash(\text{saw}/y)
\end{aligned}$$

Similarly for (27), but not for (28): the transitions for (28) get stuck, because $\lambda\ell$ in (30) needs a license and does not get it.

As in Fry's analyses, our translations of *nobody* and *anybody* integrate the scope-taking and polarity-licensing aspects of their meanings. Consequently, *nobody* must take scope *immediately* over one or more occurrences of *anybody* in order to license them. Essentially, (29)–(31) specify a finite-state machine that accepts strings of scope-taking elements in properly licensed order. It is easy to incorporate into the same system other scope-taking elements, such as *somebody* and *everybody* and their surprising interactions [19].

3.3 Quotation

A deterministic transition relation predicts wrongly that quantifier scope can never be ambiguous. In particular, the transition relation above enforces call-by-value, left-to-right evaluation and thus forces quantifiers that do not contain each other to take surface scope with respect to each other. We refine our empirical predictions using the notion of metalinguistic quotation, as expressed in operational semantics by *multistage programming* [23, inter alia].

Briefly, multistage programming makes three new constructs available in programs: *quotation*, *splice*, and *run*. Quoting an expression such as $1 + 2$, notated $\lceil 1 + 2 \rceil$, turns it into a static value, a piece of code. If x is a piece of code, then

it can be spliced into a quotation, notated $\lfloor x \rfloor$. For example, if x is $\lceil 1+2 \rceil$, then $\lceil \lfloor x \rfloor \times \lfloor x \rfloor \rceil$ is equivalent to $\lceil (1+2) \times (1+2) \rceil$. Finally, $!x$ notates running a piece of code. The transitions below illustrate.

(33)
$$\begin{aligned}
&\big(\lambda x.\, !\lceil \lfloor x \rfloor \times \lfloor x \rfloor \rceil\big)\big(\lceil 1+2 \rceil\big) \\
&\quad \rightsquigarrow\ !\lceil \lfloor \lceil 1+2 \rceil \rfloor \times \lfloor \lceil 1+2 \rceil \rfloor \rceil \\
&\quad \rightsquigarrow\ !\lceil (1+2) \times \lfloor \lceil 1+2 \rceil \rfloor \rceil \rightsquigarrow\ !\lceil (1+2) \times (1+2) \rceil \\
&\quad \rightsquigarrow\ (1+2) \times (1+2) \rightsquigarrow\ 3 \times (1+2) \rightsquigarrow\ 3 \times 3 \rightsquigarrow\ 9
\end{aligned}$$

We need to add to our transition relation the general cases of the second, third, and fourth transitions above. We omit these formal definitions, which involve augmenting evaluation contexts too.

We contend that inverse scope is an instance of multistage programming. Specifically, we propose that the sentence (24) has an inverse-scope reading because it translates not just to the program (25) but also to the multistage program

(34)
$$\text{reset!}\lceil \text{reset}\big(\overbrace{(\text{shift}\, f.\, \exists x.\, fx)}^{somebody}\backslash\overbrace{(\text{saw}}^{saw}\, /\overbrace{\lfloor \text{shift}\, g.\, \forall y.\, g\lceil y \rceil \rfloor}^{everybody}\big)\big)\rceil.$$

This latter program makes the following transitions, even under left-to-right evaluation.

$$\begin{aligned}
&\rightsquigarrow \text{reset}\big(\forall y.\, \big(\lambda y.\, \text{reset!}\lceil \text{reset}((\text{shift}\, f.\, \exists x.\, fx)\backslash(\text{saw}/\lfloor y \rfloor))\rceil\big)\lceil y \rceil\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \text{reset!}\lceil \text{reset}((\text{shift}\, f.\, \exists x.\, fx)\backslash(\text{saw}/\lfloor \lceil y \rceil \rfloor))\rceil\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \text{reset!}\lceil \text{reset}((\text{shift}\, f.\, \exists x.\, fx)\backslash(\text{saw}/y))\rceil\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \text{reset reset}((\text{shift}\, f.\, \exists x.\, fx)\backslash(\text{saw}/y))\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \text{reset reset}(\exists x.\, (\lambda x.\, \text{reset}(x\backslash(\text{saw}/y)))x)\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \text{reset reset}(\exists x.\, \text{reset}(x\backslash(\text{saw}/y)))\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \text{reset reset}(\exists x.\, x\backslash(\text{saw}/y))\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \text{reset}(\exists x.\, x\backslash(\text{saw}/y))\big) \\
&\rightsquigarrow \text{reset}\big(\forall y.\, \exists x.\, x\backslash(\text{saw}/y)\big) \rightsquigarrow\ \forall y.\, \exists x.\, x\backslash(\text{saw}/y)
\end{aligned}$$

The intuition behind this translation is that the scope of *everybody* in the inverse-scope reading of (24) is metalinguistically quoted. The program (34) may be glossed as "Everybody y is such that the sentence *Somebody saw* y is true", except it makes no sense to splice a person into a sentence, but it does make sense to splice the quotation $\lceil y \rceil$ in (34) into a sentence [17].

Several empirical advantages of this account of inverse scope, as opposed to just allowing non-left-to-right evaluation, lie in apparently noncompositional phenomena other than (but closely related to) quantification. In particular, we explain why a polarity licensor cannot take inverse scope over a polarity item [9, 15].

(35) *Anybody saw nobody.

Surface scope is unavailable for (35) simply because *anybody* must take scope (right) under its licensor. All current accounts of polarity sensitivity capture this generalization, including that sketched in Section 3.2. A more enduring puzzle is why inverse scope is also unavailable. Intuitively, our analysis of inverse scope rules out (35) because it would gloss it as "Nobody y is such that the sentence *Anybody saw y* is true" but *Anybody saw y* is not a well-formed sentence.

Formally, we hypothesize that quotation only proceeds by enclosing the translation of clauses in $\text{reset!}\lceil I_t\,\text{reset}\ldots\rceil$, where I_t is an identity function restricted to take proposition arguments only. In other words, the only transition from a program of the form $C[I_t V]$, where $C[\ ]$ is any evaluation context, is to $C[V]$ when V is a proposition (rather than a function such as $\lambda\ell.\ldots$). Replacing $\text{reset!}\lceil\text{reset}\ldots\rceil$ in (34) by $\text{reset!}\lceil I_t\,\text{reset}\ldots\rceil$ does not hamper the transitions there, because $\exists x.\, x\backslash(\text{saw}/y)$ is a proposition. In contrast, even though (35) translates successfully to the program

$$\text{(36)}\quad \text{reset!}\lceil I_t\,\text{reset}\big(\overbrace{(\text{shift}\,f.\,\lambda\ell.\,\exists x.\,fx\ell)}^{anybody}\backslash\overbrace{(\text{saw}}^{saw}/\overbrace{\lfloor\text{shift}\,g.\,\neg\exists y.\,g\lceil y\rceil\ell\rfloor}^{nobody})\big)\rceil,$$

it then gets stuck after the following transitions.

$$\begin{aligned}
&\rightsquigarrow \text{reset}\big(\neg\exists y.\,\big(\lambda y.\,\text{reset!}\lceil I_t\,\text{reset}((\text{shift}\,f.\,\lambda\ell.\,\exists x.\,fx\ell)\backslash(\text{saw}/\lfloor y\rfloor))\rceil\big)\lceil y\rceil\ell\big)\\
&\rightsquigarrow \text{reset}\big(\neg\exists y.\,\big(\text{reset!}\lceil I_t\,\text{reset}((\text{shift}\,f.\,\lambda\ell.\,\exists x.\,fx\ell)\backslash(\text{saw}/\lfloor\lceil y\rceil\rfloor))\rceil\big)\ell\big)\\
&\rightsquigarrow \text{reset}\big(\neg\exists y.\,\big(\text{reset!}\lceil I_t\,\text{reset}((\text{shift}\,f.\,\lambda\ell.\,\exists x.\,fx\ell)\backslash(\text{saw}/y))\rceil\big)\ell\big)\\
&\rightsquigarrow \text{reset}\big(\neg\exists y.\,\big(\text{reset}(I_t\,\text{reset}((\text{shift}\,f.\,\lambda\ell.\,\exists x.\,fx\ell)\backslash(\text{saw}/y)))\big)\ell\big)\\
&\rightsquigarrow \text{reset}\big(\neg\exists y.\,\big(\text{reset}(I_t\,\text{reset}(\lambda\ell.\,\exists x.\,(\lambda x.\,\text{reset}(x\backslash(\text{saw}/y)))x\ell)\big)\big)\ell\big)\\
&\rightsquigarrow \text{reset}\big(\neg\exists y.\,\big(\text{reset}(I_t(\lambda\ell.\,\exists x.\,(\lambda x.\,\text{reset}(x\backslash(\text{saw}/y)))x\ell)\big)\big)\ell\big)
\end{aligned}$$

The scope of *nobody*, namely *anybody saw* __, is a function rather than a proposition, so the intervening I_t blocks licensing. In general, a polarity item must be evaluated after its licensor, because a quantifier can take inverse scope only over a proposition.

Our operational semantics of metalinguistic quotation, like Barker and Shan's analyses of polarity sensitivity [1, 18], thus joins a syntactic notion of order to a semantic notion of scope in an account of polarity—as desired [9, 15].

3.4 Other linguistic phenomena

Quantification and polarity sensitivity are just two out of many apparently non-compositional phenomena in natural language, which we term *linguistic side effects* [20]. Two other linguistic side effects are anaphora and interrogation. As the term suggests, each effect finds a natural treatment in operational semantics. For example, it is an old idea to treat anaphora as mediated by a record of referents introduced so far in the discourse [10–12]. We can express this idea in an operational semantics either by adding the record to the state as a separate component, as sketched in Section 1, or by integrating the record into

the evolving program as it rewrites [7]. Another old idea is that wh-words take scope to circumscribe how an asker and an answerer may interact [13]. Our use of delimited control extends to this instance of scope taking.

The payoff of recasting these old ideas in a general operational framework goes beyond conceptual clarity and notational simplicity. Our notions of *context* and *order* apply uniformly to all linguistic phenomena and make borne-out predictions. For example, left-to-right evaluation explains not just the interaction between quantification and polarity sensitivity but also crossover in binding and superiority in questions [1, 22].

4 Conclusion

We have shown how an operational semantics for delimited control and multi-stage programming in natural language helps explain inverse scope, polarity sensitivity, and their interaction. The foundations and applications of our meta-language remain open for investigation.

Our approach extends dynamics semantics from the intersentential level to the intrasentential level, where side effects are incurred not only by sentences (*A man walks in the park*) but also by other phrases (*nobody*). Thus discourse context is not a sequence of utterances in linear order but a tree of constituents in evaluation order. This view unifies many aspects of human language—syntactic derivation, semantic evaluation, and pragmatic update—as compatible transitions among a single set of states. A tight link between operational and denotational semantics [2] promises to strengthen the connection between the views of language as product and language as action [24].

References

[1] Barker, Chris, and Chung-chieh Shan. 2006. Types as graphs: Continuations in type logical grammar. *Journal of Logic, Language and Information* 15(4): 331–370.

[2] 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`.

[3] ———. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.

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

[5] 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.

[6] ———. 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.

[7] Felleisen, Matthias, and Daniel P. Friedman. 1989. A syntactic theory of sequential state. *Theoretical Computer Science* 69(3):243–287.

[8] Fry, John. 1997. Negative polarity licensing at the syntax-semantics interface. In *Proceedings of the 35th annual meeting of the Association for Computational Linguistics and 8th conference of the European chapter of the Association for Computational Linguistics*, ed. Philip R. Cohen and Wolfgang Wahlster, 144–150. San Francisco: Morgan Kaufmann.

[9] ———. 1999. Proof nets and negative polarity licensing. In *Semantics and syntax in Lexical Functional Grammar: The resource logic approach*, ed. Mary Dalrymple, chap. 3, 91–116. Cambridge: MIT Press.

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

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

[12] 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.

[13] Karttunen, Lauri. 1977. Syntax and semantics of questions. *Linguistics and Philosophy* 1(1):3–44.

[14] Kiselyov, Oleg, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP '06: Proceedings of the ACM international conference on functional programming*, 26–37. New York: ACM Press.

[15] Ladusaw, William A. 1979. Polarity sensitivity as inherent scope relations. Ph.D. thesis, Department of Linguistics, University of Massachusetts. Reprinted by New York: Garland, 1980.

[16] Plotkin, Gordon D. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Department of Computer Science, University of Aarhus. Revised version submitted to *Journal of Logic and Algebraic Programming*.

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

[18] Shan, Chung-chieh. 2004. Delimited continuations in natural language: Quantification and polarity sensitivity. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke, 55–64. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.

[19] ———. 2004. Polarity sensitivity and evaluation order in type-logical grammar. In *Proceedings of the 2004 human language technology conference of the North American chapter of the Association for Computational Linguistics*, ed. Susan Dumais, Daniel Marcu, and Salim Roukos, vol. 2, 129–132. Somerset, NJ: Association for Computational Linguistics.

[20] ———. 2005. Linguistic side effects. Ph.D. thesis, Harvard University.

[21] ———. 2007. Linguistic side effects. In *Direct compositionality*, ed. Chris Barker and Pauline Jacobson, 132–163. New York: Oxford University Press.
[22] Shan, Chung-chieh, and Chris Barker. 2006. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy* 29(1):91–134.
[23] Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 26–37. New York: ACM Press.
[24] Trueswell, John C., and Michael K. Tanenhaus, eds. 2005. *Approaches to studying world-situated language use: Bridging the language-as-product and language-as-action traditions.* Cambridge: MIT Press.