Composing meanings as programs

Chung-chieh Shan
Rutgers University

European Summer School in Logic, Language and Information
4–8 August 2008

Contents

1 Inverse scope as metalinguistic quotation in operational semantics **2**

Chung-chieh Shan. In *New frontiers in artificial intelligence, JSAI 2007 conference and workshops, revised selected papers*, ed. Ken Satoh, Akihiro Inokuchi, Katashi Nagao, and Takahiro Kawamura, 123–134. Lecture notes in computer science 4914, Springer, 2008.

2 On evaluation contexts, continuations, and the rest of the computation **8**

Olivier Danvy. In *Proceedings of the 4th continuations workshop*, ed. Hayo Thielecke. Technical Report CSR-04-1, School of Computer Science, University of Birmingham, 2004.

3 Donkey anaphora is in-scope binding **19**

Chris Barker and Chung-chieh Shan. *Semantics and Pragmatics* 1(1): 1–42, 2008.

4 Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages **40**

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. In *Proceedings of the 5th ASIAN symposium on programming languages and systems*, ed. Zhong Shao, 222–238. Lecture notes in computer science 4807, Springer, 2007.

5 A gentle introduction to multi-stage programming **48**

Walid Taha. In *Domain-specific program generation, international seminar, 2003, revised papers*, ed. Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, 30–50. Lecture notes in computer science 3016, Springer, 2004.

6 Interpreting quotations **59**

Chung-chieh Shan. Draft manuscript, 2008.

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 [1]. 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

(1) $$2 \to n;\ n \cdot 3$$

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 [2]. 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

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

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

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

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

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

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.

(5) $$E \to x \qquad E \to \lambda x.\, E \qquad E \to EE$$

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

(6) $$E \to \lambda x.\, E \to \lambda x.\, EE \to \lambda x.\, Ex \to \lambda x.\, xx$$

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.

(7) $$V \to x \qquad\qquad V \to \lambda x.\,E$$

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 [3]. The notation $C[\dots]$ means to replace the gap $[\,]$ in the context $C[\,]$ by an expression or another context.

(8) $$C[\,] \to [\,] \qquad C[\,] \to C\big[[\,]E\big] \qquad C[\,] \to C\big[V[\,]\big]$$

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 $\lambda x.\,x$.)

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

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

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

or for short,

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

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

(12) $$I\big((II)(II)\big) \rightsquigarrow I\big(I(II)\big).$$

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:

(13) $$I\big(I(II)\big) \rightsquigarrow I(II) \rightsquigarrow II \rightsquigarrow I.$$

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* [3, 4], which lets an expression manipulate its context. To illustrate, we add 's delimited-control operators *shift* and *reset* [5, 6, 7] 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:

(14) $$1+2\cdot 3 \rightsquigarrow 1+6 \rightsquigarrow 7,$$

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

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 $1+3\cdot 2\cdot 2\cdot 5$.

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

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.

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

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.

$$(18) \qquad E \to \text{reset}\, E \qquad\qquad E \to \text{shift}\, f.\, E$$

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.

$$(19) \qquad C[\,] \to C\big[\text{reset}[\,]\big]$$

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:

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

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

or for short,

$$(22) \qquad C\big[\text{reset}\, V\big] \rightsquigarrow C\big[V\big],$$

$$(23) \qquad C\big[\text{reset}\, D[\text{shift}\, f.\, E]\big] \rightsquigarrow C\big[\text{reset}\, E\,\{f \mapsto \lambda x.\,\text{reset}\, D[x]\}\big].$$

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)

$$(25) \qquad \text{reset}\Big(\overbrace{(\text{shift}\, f.\, \exists x.\, fx)}^{\textit{somebody}}{}^{<}\big(\overbrace{\text{saw}}^{\textit{saw}}{}^{>}\overbrace{(\text{shift}\, g.\, \forall y.\, gy)}^{\textit{everybody}}\big)\Big),$$

where $^{<}$ and $^{>}$ indicate backward and forward function application. The occurrences of shift in this translation arise from the lexical entries for the quantifiers *somebody* and *everybody*, whereas the occurrence of reset is freely available at every clause boundary. This program then makes the following transitions to yield the surface-scope reading of (25).

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

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 [8–11] 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 [12, 13], 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

$$(29) \qquad \text{shift}\, f.\, \neg\exists x.\, fx\ell,$$

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).

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

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 the positive polarity item *somebody* and its surprising interaction with *everybody* [15–17]: surface scope is possible in (33) but not the simpler sentence (34).

(33) Nobody introduced everybody to somebody.

(34) Nobody introduced Alice to somebody.

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* [18, 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.

(35)
$$\begin{aligned}
&\big(\lambda x.\, !\lceil \lfloor x \rfloor \times \lfloor x \rfloor \rceil\big)\big(\lceil 1+2 \rceil\big)\\
&\quad \rightsquigarrow\, !\lceil \lfloor \lceil 1+2 \rceil \rfloor \times \lfloor \lceil 1+2 \rceil \rfloor \rceil\\
&\quad \rightsquigarrow\, !\lceil (1+2) \times \lfloor \lceil 1+2 \rceil \rfloor \rceil \rightsquigarrow\, !\lceil (1+2) \times (1+2) \rceil\\
&\quad \rightsquigarrow (1+2) \times (1+2) \rightsquigarrow 3 \times (1+2) \rightsquigarrow 3 \times 3 \rightsquigarrow 9
\end{aligned}$$

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

$$\text{reset}!\Big\lceil \text{reset}\Big(\big(\overbrace{\text{shift}\, f.\, \exists x.\, fx}^{somebody}\big)^{<}\big(\overbrace{\text{saw}}^{saw}{}^{>}\big\lfloor \overbrace{\text{shift}\, g.\, \forall y.\, g\lceil y \rceil}^{everybody}\big\rfloor\big)\Big)\Big\rceil. \tag{36}$$

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

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

The intuition behind this translation is that the scope of *everybody* in the inverse-scope reading of (24) is metalinguistically quoted. The program (36) 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 (36) into a sentence [19].

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 [13, 20].

(37) *Anybody saw nobody.

Surface scope is unavailable for (37) 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 (37) 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 $\lambda\ell.\ldots$). We hypothesize I_t not because the operational semantics forces us to, but to express the intuition (some might say stipulation) that quotation applies to propositions only.

Replacing $\text{reset!}\lceil\text{reset}\ldots\rceil$ in (36) by $\text{reset!}\lceil I_t\,\text{reset}\ldots\rceil$ does not hamper the transitions there, because $\exists x.\, x^{<}(\text{saw}^{>}y)$ is a proposition. In contrast, even though (37) translates successfully to the program

$$\text{(38)}\quad \text{reset!}\Big\lceil I_t\,\text{reset}\Big(\big(\overbrace{\text{shift}\, f.\, \lambda\ell.\, \exists x.\, fx\ell}^{\textit{anybody}}\big)^{<}\big(\overbrace{\text{saw}}^{\textit{saw}}{}^{>}\overbrace{\lfloor \text{shift}\, g.\, \neg\exists y.\, g\lceil y\rceil\ell\rfloor}^{\textit{nobody}}\big)\Big)\Big\rceil,$$

it then gets stuck after the following transitions.

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

The scope of *nobody*, namely *anybody saw* __, is a function rather than a proposition, so the intervening I_t blocks licensing. (The I_t would block licensing even if we quote the license ℓ—that is, even if we replace the last ℓ in (38) by $\lceil\ell\rceil$.) 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 [8, 21], thus joins a syntactic notion of order to a semantic notion of scope in an account of polarity—as desired [13, 20]. The general strategy is to proliferate clause types: a clause in the analysis above may denote not a proposition but a function from ℓ to propositions. We can generalize this strategy to more clause types in order to account for additional quantifiers and polarity items, such as in English [15], Dutch, Greek, Italian [22], and Hungarian [23].

3.4 Other linguistic phenomena

Quantification and polarity sensitivity are just two out of many apparently noncompositional phenomena in natural language, which we term *linguistic side effects* [11]. 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 [24–26]. 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 [27]. Another old idea is that wh-words take scope to circumscribe how an asker and an answerer may interact [28]. 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 [8, 9].

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. We are actively investigating the foundations and applications of our metalanguage, where many open issues remain. In particular, there is currently no type system or denotational semantics on the market that soundly combines delimited control and quotation, so we have no way to understand type-logically or statically what it means for a program such as (38) to get stuck, much as we would like to.

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 [5] promises to strengthen the connection between the views of language as product and language as action [1].

References

[1] Trueswell, J.C., Tanenhaus, M.K., eds.: Approaches to Studying World-Situated Language Use: Bridging the Language-as-Product and Language-as-Action Traditions. MIT Press, Cambridge (2005)

[2] Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, University of Aarhus (1981) Revised version submitted to *Journal of Logic and Algebraic Programming*.

[3] Felleisen, M.: The Calculi of λ_v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages. PhD thesis, Computer Science Department, Indiana University (1987) Also as Tech. Rep. 226.

[4] Felleisen, M.: The theory and practice of first-class prompts. In: POPL '88: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, New York, ACM Press (1988) 180–190

[5] Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Denmark (1989) `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`.

[6] Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, New York, ACM Press (1990) 151–160

[7] Danvy, O., Filinski, A.: Representing control: A study of the CPS transformation. Mathematical Structures in Computer Science **2** (1992) 361–391

[8] Barker, C., Shan, C.c.: Types as graphs: Continuations in type logical grammar. Journal of Logic, Language and Information **15** (2006) 331–370

[9] Shan, C.c., Barker, C.: Explaining crossover and superiority as left-to-right evaluation. Linguistics and Philosophy **29** (2006) 91–134

[10] Shan, C.c.: Linguistic side effects. In Barker, C., Jacobson, P., eds.: Direct Compositionality, New York, Oxford University Press (2007) 132–163

[11] Shan, C.c.: Linguistic Side Effects. PhD thesis, Harvard University (2005)

[12] Fry, J.: Negative polarity licensing at the syntax-semantics interface. In Cohen, P.R., Wahlster, W., eds.: 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, San Francisco, Morgan Kaufmann (1997) 144–150

[13] Fry, J.: Proof nets and negative polarity licensing. In Dalrymple, M., ed.: Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach. MIT Press, Cambridge (1999) 91–116

[14] Kiselyov, O., Shan, C.c., Sabry, A.: Delimited dynamic binding. In: ICFP '06: Proceedings of the ACM International Conference on Functional Programming, New York, ACM Press (2006) 26–37

[15] Shan, C.c.: Polarity sensitivity and evaluation order in type-logical grammar. In Dumais, S., Marcu, D., Roukos, S., eds.: Proceedings of the 2004 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics. Volume 2., Somerset, NJ, Association for Computational Linguistics (2004) 129–132

[16] Kroch, A.S.: The Semantics of Scope in English. PhD thesis, Massachusetts Institute of Technology (1974) Reprinted by New York: Garland, 1979.

[17] Szabolcsi, A.: Positive polarity—negative polarity. Natural Language and Linguistic Theory **22** (2004) 409–452

[18] Taha, W., Nielsen, M.F.: Environment classifiers. In: POPL '03: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, New York, ACM Press (2003) 26–37

[19] Quine, W.V.O.: Word and Object. MIT Press, Cambridge (1960)

[20] Ladusaw, W.A.: Polarity Sensitivity as Inherent Scope Relations. PhD thesis, Department of Linguistics, University of Massachusetts (1979) Reprinted by New York: Garland, 1980.

[21] Shan, C.c.: Delimited continuations in natural language: Quantification and polarity sensitivity. In Thielecke, H., ed.: CW'04: Proceedings of the 4th ACM SIGPLAN Continuations Workshop. Number CSR-04-1 in Tech. Rep., School of Computer Science, University of Birmingham (2004) 55–64

[22] Bernardi, R.: Reasoning with Polarity in Categorial Type Logic. PhD thesis, Utrecht Institute of Linguistics (OTS), Utrecht University (2002)

[23] Bernardi, R., Szabolcsi, A.: Partially ordered categories: Optionality, scope, and licensing. `http://ling.auf.net/lingBuzz/000372` (2006)

[24] Kamp, H.: A theory of truth and semantic representation. In Groenendijk, J.A.G., Janssen, T.M.V., Stokhof, M.B.J., eds.: Formal Methods in the Study of Language: Proceedings of the 3rd Amsterdam Colloquium, Amsterdam, Mathematisch Centrum (1981) 277–322

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

[26] Heim, I.: The Semantics of Definite and Indefinite Noun Phrases. PhD thesis, Department of Linguistics, University of Massachusetts (1982)

[27] Felleisen, M., Friedman, D.P.: A syntactic theory of sequential state. Theoretical Computer Science **69** (1989) 243–287

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

On Evaluation Contexts, Continuations, and the Rest of the Computation

Olivier Danvy
BRICS*
Department of Computer Science
University of Aarhus†

Abstract

Continuations are variously understood as representations of the current evaluation context and as representations of the rest of the computation, but these understandings contradict each other: plugging an expression in a context yields a new expression whereas sending an intermediate result to a continuation yields the final answer. We show that continuations-as-evaluation-contexts are the defunctionalized representation of the continuation of a single-step reduction function and that continuations-as-the-rest-of-the-computation are the continuation of an evaluation function. Furthermore, we show that defunctionalizing the continuation of an evaluator gives rise to the same evaluation contexts as in the single-step reducer. The only difference is how these evaluation contexts are interpreted: a 'plug' interpretation yields one-step reduction, whereas a 'refocus' interpretation yields evaluation.

We then present a constructive corollary of Reynolds's historical warning about depending on the evaluation order of a meta-language for an interpreter: The two best-known abstract machines for the λ-calculus, Krivine's machine and Felleisen et al.'s CEK machine, are in fact the call-by-name and call-by-value counterparts of the *same* (evaluation-order dependent) interpreter for the λ-calculus.

1 Introduction

The notion of continuation is ubiquitous in many different areas of computer science, including logic, constructive mathematics, programming languages, and programming. Nevertheless, continuations are a remarkably elusive, even mystifying, notion. They pop up virtually everywhere as a uniform solution to control-related problems, and it seems that no two alternative solutions to these problems are alike. Worse, no particular effort seems to have been devoted to connecting these alternative solutions to the solutions based on continuations and from there, to transpose these alternative solutions to other domains.

* Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

† Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. Email: `danvy@brics.dk`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CW'04 January 17, 2004, Venice, Italy.
Copyright 2004 ACM baz ...$5.00

1.1 Continuations, informally

What is a continuation? To some, it is the representation of an evaluation context, i.e., an expression with a hole; plugging an expression into this hole yields a new expression. To others, a continuation is a representation of the rest of the computation; sending it an intermediate result yields the final result of the overall computation. These two notions are plausible and even widespread (the latter one is actually the original definition [63]), but they are incompatible with each other. In the former case, a continuation expects an expression and returns another expression. In the latter case, a continuation expects a value and returns a final result.

The primary goal of this article is to reconcile these two common, but contradictory, understandings of continuations as representations of the current context and as representations of the rest of the computation.

1.2 Continuations, authoritatively

When they are mentioned at all, continuations are presented with considerable variations in textbook and lecture notes. In *Concepts in Programming Languages* [47], Mitchell briefly defines a continuation as a function representing the remaining program to evaluate; he mentions continuation-passing style as a way to obtain tail recursion. In *Programming Languages: Theory and Practice* [40], Harper summarily defines a continuation as a control stack and argues that a formal semantics is much clearer; he mentions continuation-passing style as a way to "roll one's own" continuation. In *Compiling with Continuations* [5], Appel defines a continuation as a function that expresses what to do next; he then makes a substantial use of continuation-passing style. In *Essentials of Programming Languages* [34], Friedman, Wand, and Haynes define a continuation as an abstraction of the control context; they dedicate two chapters to continuation-passing interpreters and transforming programs into continuation-passing style. In *Programming Languages and Lambda Calculi* [28], Felleisen and Flatt define a continuation as an inside-out evaluation context in an abstract machine; they do not consider continuation-passing style. In *Lisp in Small Pieces* [53], Queinnec defines a continuation as a representation of all that remains to compute; he mentions contexts as an alternative representation of continuations.

A secondary goal of this article is to unify these common, but distinct, representations of continuations. Our thesis is that Reynolds's defunctionalization provides the key to this unification, in the sense that control stacks and evaluation contexts are defunctionalized continuations.

1.3 Prerequisites

We expect a passing familiarity with functional programming (ML), and we build on the notions of evaluators, abstract machines, CPS transformation, defunctionalization, and syntactic theories:

Evaluation functions: An evaluator is a compositional function mapping an abstract-syntax tree to an expressible value, if there is one; it implements a denotational semantics [58].

Abstract machines: An abstract machine is a transition function over computational states; it implements an operational semantics [52].

CPS transformation: A program is transformed into continuation-passing style (CPS) by naming all of its intermediate results, sequentializing their computation, and introducing continuations. Each CPS transformation encodes an evaluation order [20, 41, 51, 57, 62].

Defunctionalization: A program is defunctionalized by replacing each of its function spaces by a first-order data type and a first-order apply function [56]. Each data type enumerates all the function abstractions that may give rise to inhabitants of the corresponding function space [7, 8, 13, 21, 49, 56, 65].

A particular case of defunctionalization is closure conversion: in an evaluator, closure conversion amounts to replacing each of the function spaces in expressible and denotable values by a tuple, and inlining the corresponding apply function [46, 56]. (Other styles of closure conversion exist, though [6].)

Syntactic theories: A syntactic theory provides a reduction relation on expressions by defining syntax, values, evaluation contexts, and redexes [26, 28, 70]. For example, a syntactic theory for arithmetic expressions is specified as follows.

Syntax: $e ::= n \mid e+e$

Values: n

Redexes: $n+n'$

Evaluation contexts: $E ::= [\,] \mid E[n+[\,]] \mid E[[\,]+e]$

Plugging an expression e into a context E:

$$\begin{array}{rcl} plug([\,],e) & = & e \\ plug(E[n+[\,]],e) & = & plug(E,n+e) \\ plug(E[[\,]+e'],e) & = & plug(E,e+e') \end{array}$$

Reduction relation: $E[n+n'] \to E[n'']$, where n'' is the sum of n and n'.

These definitions satisfy a "unique decomposition" lemma [70]: any expression e that is not a value can be uniquely decomposed into an evaluation context E and a redex $n+n'$ such that $e = plug(E,n+n')$.

From syntactic theory to abstract machine: Nielsen and the author have established the conditions under which one can deforest an evaluation function when it is defined as the transitive closure of one-step reduction in a syntactic theory [22]. At each step, a term is decomposed into an evaluation context and a redex, the redex is contracted, and the contractum is plugged into the evaluation context. Deforesting such an evaluation function makes it possible to avoid the construction of intermediate expressions. Our key point is to construct a "refocus" function that makes it possible to replace the decompose-contract-plug-decompose-contract-plug-... loop by an initial decomposition followed by a contract-refocus-contract-refocus-... loop. The result is an abstract machine.

For example, here is the refocus function corresponding to the syntactic theory just above:

$$\begin{array}{rcl} refocus([\,],n) & = & n \\ refocus(E[n'+[\,]],n) & = & refocus(E,n'+n) \\ refocus(E[[\,]+e],n) & = & decompose(e,E[n+[\,]]) \end{array}$$

where *decompose* decomposes a computation into an evaluation context and a redex.

1.4 Overview

The rest of this article is organized as follows. We first investigate continuations as evaluation contexts and continuations as the rest of the computation; to this end, we revisit the simple example of arithmetic expressions above (Section 2). We then consider the λ-calculus (Section 3) and analyze further consequences (Section 4).

2 A simple example: arithmetic expressions

To investigate continuations as evaluation contexts and continuations as the rest of the computation, we go through the simple exercise of writing a one-step reduction function and then an evaluator for arithmetic expressions. We write each of them in direct style, and we successively CPS-transform them and then defunctionalize their continuations.

Our arithmetic expressions are minimal: they consist of literals and additions.

```
datatype exp = VALUE of value
             | COMP of comp
and    value = LIT of int
and     comp = ADD of exp * exp
```

Literals are the only values and additions are the only computations.

2.1 A one-step reduction function

We write the one-step reduction function by recursive descent, using the recursive calls to reach the left-most-innermost redex, and constructing the reduced expression at return time:

```
(*  reduce1 : comp -> exp  *)
fun reduce1 (ADD (VALUE (LIT n1), VALUE (LIT n2)))
    = VALUE (LIT (n1 + n2))
  | reduce1 (ADD (VALUE v1, COMP c2))
    = COMP (ADD (VALUE v1, reduce1 c2))
  | reduce1 (ADD (COMP c1, e2))
    = COMP (ADD (reduce1 c1, e2))
```

We then CPS-transform `reduce1`:

```
(*  reduce1c : comp * (exp -> 'a) -> 'a  *)
fun reduce1c (ADD (VALUE (LIT n1), VALUE (LIT n2)), k)
    = k (VALUE (LIT (n1 + n2)))
  | reduce1c (ADD (VALUE v1, COMP c2), k)
    = reduce1c (c2, fn e2 => k (COMP (ADD (VALUE v1, e2))))
  | reduce1c (ADD (COMP c1, e2), k)
    = reduce1c (c1, fn e1 => k (COMP (ADD (e1, e2))))
```

Finally, we defunctionalize the continuations in `reduce1c`. We assume an initial continuation that is the identity function, and therefore the polymorphic type variable in the type of `reduce1c` is specialized to `exp`. Three functional abstractions can build inhabitants in the function space `exp -> exp`. The first is the initial continuation and it has no free variables. The second is the continuation in the second clause, and it has `v1` and `k` as free variables. The third is the continuation in the third clause, and it has `e2` and `k` as free variables. The data type of defunctionalized continuations has thus three constructors.

```
datatype cont = CONT0
              | CONT1 of value * cont
              | CONT2 of exp * cont

(*  apply : cont * exp -> exp  *)
fun apply (CONT0, e)
    = e
  | apply (CONT1 (v1, k), e2)
    = apply (k, COMP (ADD (VALUE v1, e2)))
  | apply (CONT2 (e2, k), e1)
    = apply (k, COMP (ADD (e1, e2)))

(*  reduce1cd : comp * cont -> exp  *)
fun reduce1cd (ADD (VALUE (LIT n1), VALUE (LIT n2)), k)
    = apply (k, VALUE (LIT (n1 + n2)))
  | reduce1cd (ADD (VALUE v1, COMP c2), k)
    = reduce1cd (c2, CONT1 (v1, k))
  | reduce1cd (ADD (COMP c1, e2), k)
    = reduce1cd (c1, CONT2 (e2, k))
```

We observe that the data type `cont` is isomorphic to the data type of evaluation contexts for arithmetic expressions, and that its apply function coincides with the corresponding plug function. *Evaluation contexts, together with their plug function, are therefore a representation of the continuation of a one-step reduction function.*

2.2 An evaluation function

We write an evaluation function by recursive descent:

```
(*  eval : exp -> int  *)
fun eval (VALUE (LIT n))
    = n
  | eval (COMP (ADD (e1, e2)))
    = (eval e1) + (eval e2)

(*  main : exp -> int  *)
fun main e
    = eval e
```

We then CPS-transform `eval`:

```
(*  evalc : exp * (int -> 'a) -> 'a  *)
fun evalc (VALUE (LIT n), k)
    = k n
  | evalc (COMP (ADD (e1, e2)), k)
    = evalc (e1,
             fn n1 => evalc (e2,
                             fn n2 => k (n1 + n2)))

(*  main : exp -> int  *)
fun main e
    = eval (e, fn n => n)
```

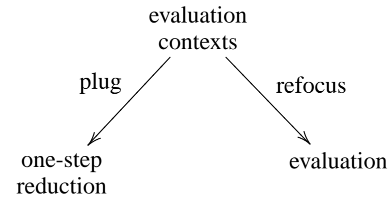
Finally, we defunctionalize the continuations in `evalc`. The initial continuation is the identity function and therefore the polymorphic type variable in the type of `evalc` is specialized to `int`. Three functional abstractions can build inhabitants in the function space `int -> int`. The first is the initial continuation and it has no free variables. The second is the inner continuation in the `ADD` clause, and it has `n1` and `k` as free variables. The third is the outer continuation in the `ADD` clause, and it has `e2` and `k` as free variables. The data type of defunctionalized continuations thus has three constructors. Due to the recursive call to `evalc` in the outer continuation, the apply function of defunctionalized continuations and the defunctionalized version of `evalc` are mutually recursive:

```
datatype cont = CONT0
              | CONT1 of int * cont
              | CONT2 of exp * cont

(*  apply : cont * int -> int  *)
fun apply (CONT0, n)
    = n
  | apply (CONT1 (n1, k), n2)
    = apply (k, n1 + n2)
  | apply (CONT2 (e2, k), n1)
    = evalcd (e2, CONT1 (n1, k))

(*  evalcd : exp * cont -> int  *)
and evalcd (VALUE (LIT n), k)
    = apply (k, n)
  | evalcd (COMP (ADD (e1, e2)), k)
    = evalcd (e1, CONT2 (e2, k))

(*  main : exp -> int  *)
fun main e
    = eval (e, CONT0)
```

We observe that the data type `cont` is isomorphic to the data type of evaluation contexts for arithmetic expressions, and that its apply function coincides with the corresponding refocus function. *Evaluation contexts, together with their refocus function, are therefore a representation of the continuation of an evaluation function.*

2.3 Conclusion

Continuations have two sides: they can represent the context for one-step reduction and they can represent the rest of the computation for evaluation. Common to both sides is the notion of evaluation context:



- Evaluation contexts, together with the plug interpretation, are the defunctionalized representation of the continuation of a one-step reducer.
- Evaluation contexts, together with the refocus interpretation, are the defunctionalized representation of the continuation of an evaluator.

Identifying these two representations of evaluation contexts makes it possible to reconcile the two common—but contradictory—understandings of continuations as representations of the current context and as representations of the rest of the computation.

Evaluation contexts were first proposed in Felleisen's PhD thesis [26] and since then they have had a clear impact in the formal study of programming languages. Yet they have never before been formally connected with the continuation of a one-step reduction function or with the continuation of an evaluation function.

It takes some skill to define evaluation contexts. Until the unique-decomposition lemma is proven, one is never sure whether the enumeration is complete and whether it is not somehow redundant. In contrast, the characterization of evaluation contexts as a defunctionalized continuation in a recursive descent to locate the next redex provides both a guideline and a security. Also, the unique-decomposition lemma holds as a corollary when one starts from a compositional recursive descent.

```
structure Eval0
= struct
    datatype expval = FUNCT of denval -> expval
    withtype denval = expval

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = List.nth (e, n)
      | eval (ABS t, e)
        = FUNCT (fn v => eval (t, v :: e))
      | eval (APP (t0, t1), e)
        = let val (FUNCT f) = eval (t0, e)
          in f (eval (t1, e))
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

Figure 1. Canonical evaluation-order dependent evaluator

```
structure Eval1
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = expval

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = List.nth (e, n)
      | eval (ABS t, e)
        = FUNCT (t, e)
      | eval (APP (t0, t1), e)
        = let val (FUNCT (t', e')) = eval (t0, e)
          in eval (t', (eval (t1, e)) :: e')
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

Figure 2. Evaluator of Figure 1, closure-converted

3 A constructive corollary of Reynolds's evaluation-order dependence

In earlier work, the author and his students have observed that a defunctionalized CPS program implements an abstract machine [2, 3, 4, 10, 11, 18]. In particular, we have found that Krivine's abstract machine [15, 39, 45] is the defunctionalized and continuation-passing counterpart of a closure-converted call-by-value evaluator for the λ-calculus and that Felleisen et al.'s CEK machine [28, 29, 33] is the defunctionalized and continuation-passing counterpart of a closure-converted call-by-name evaluator for the λ-calculus [2].

The goal of this section is to show that Krivine's abstract machine and the CEK machine can in fact be derived from the same evaluator for the λ-calculus. This evaluator is the most canonical one for the λ-calculus: it is in direct style, higher-order, compositional, and with an environment. As pointed by Reynolds [56], it is also evaluation-order dependent: if the evaluation order of the defining language is call by name (resp. call by value), the evaluation order of the defined language is also call by name (resp. call by value). We specify this evaluation order with the corresponding CPS transformation:

- Our implementation of the abstract syntax of the λ-calculus is as follows:

  ```
  datatype term = IND of int    (* de Bruijn index *)
                | ABS of term
                | APP of term * term
  ```

  Variables are represented by their lexical offset (i.e., their de Bruijn index).

- Figure 1 displays an evaluation-order dependent evaluator in the concrete syntax of Standard ML. This evaluator is compositional (all recursive calls on the right side of the equal sign are made over proper sub-parts of the terms on the left side) and higher order (the domain of expressible values is a function space), with an environment (a list of denotable values).

- Figure 2 displays a first-order counterpart of the evaluator of Figure 1, again in the syntax of Standard ML. This evaluator was obtained by in-place defunctionalization of the expressible values, i.e., closure conversion [46, 56].

```
structure Eval1n
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = (expval -> expval) -> expval

    (*  eval : term * denval list * (expval -> expval)  *)
    (*         -> expval                                *)
    fun eval (IND n, e, k)
        = List.nth (e, n) k
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn (FUNCT (t', e')) =>
          eval (t', (fn k' => eval (t1, e, k')) :: e', k))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

Figure 3. Call-by-name CPS counterpart of Figure 2

```
structure Eval1v
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = expval

    (*  eval : term * denval list * (expval -> expval)  *)
    (*         -> expval                                *)
    fun eval (IND n, e, k)
        = k (List.nth (e, n))
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn (FUNCT (t', e')) =>
          eval (t1, e, fn v1 =>
          eval (t', v1 :: e', k)))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

Figure 4. Call-by-value CPS counterpart of Figure 2

```
structure Eval1nd
= struct
    datatype expval = FUNCT of term * denval list
         and denval = THUNK of term * denval list

    datatype cont = CONT0
                  | CONT1 of term * denval list * cont

    (*  eval : term * denval list * cont -> expval  *)
    fun eval (IND n, e, k)
        = let val (THUNK (t', e')) = List.nth (e, n)
          in eval (t', e', k)
          end
      | eval (ABS t', e', CONT1 (t1, e, k))
        = eval (t', (THUNK (t1, e)) :: e', k)
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, CONT1 (t1, e, k))
      | eval (ABS t, e, CONT0)
        = FUNCT (t, e)

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, CONT0)
  end
```

Figure 5. Defunctionalized counterpart of Figure 3

```
structure Eval1vd
= struct
    datatype expval = FUNCT of term * denval list
    withtype denval = expval

    datatype cont = CONT0
                  | CONT1 of denval * cont
                  | CONT2 of term * denval list * cont

    (*  eval : term * denval list * cont -> expval  *)
    fun eval (IND n, e, k)
        = apply (k, List.nth (e, n))
      | eval (ABS t, e, k)
        = apply (k, FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, CONT2 (t1, e, k))
    and apply (CONT2 (t1, e, k), v0)
        = eval (t1, e, CONT1 (v0, k))
      | apply (CONT1 (FUNCT (t', e'), k), v1)
        = eval (t', v1 :: e', k)
      | apply (CONT0, v)
        = v

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, CONT0)
  end
```

Figure 7. Defunctionalized counterpart of Figure 4

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [t, e]$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t, nil, nil\rangle$
$\langle n, e, s\rangle$	$\Rightarrow$	$\langle t', e', s\rangle$ where $nth(e, n) = [t', e']$
$\langle \lambda t', e', [t_1, e] :: s\rangle$	$\Rightarrow$	$\langle t', [t_1, e] :: e', s\rangle$
$\langle t_0\, t_1, e, s\rangle$	$\Rightarrow$	$\langle t_0, e, [t_1, e] :: s\rangle$
$\langle \lambda t, e, nil\rangle$	$\Rightarrow$	$[t, e]$

The abstract machine operates on triples consisting of a term, an environment, and a stack of expressible values.

Each line in the table matches a clause in Figure 5.

Figure 6. Krivine's abstract machine

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [t, e]$
- Evaluation contexts:

$$k ::= \mathtt{CONT0} \mid \mathtt{CONT1}(v, k) \mid \mathtt{CONT2}(t, e, k)$$

- Initial transition, transition rules, and final transition:

t	$\Rightarrow_{init}$	$\langle t, nil, \mathtt{CONT0}\rangle$
$\langle n, e, k\rangle$	$\Rightarrow_{eval}$	$\langle k, v\rangle$ where $nth(e, n) = v$
$\langle \lambda t, e, k\rangle$	$\Rightarrow_{eval}$	$\langle k, [t, e]\rangle$
$\langle t_0\, t_1, e, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0, e, \mathtt{CONT2}(t_1, e, k)\rangle$
$\langle \mathtt{CONT2}(t_1, e, k), v_0\rangle$	$\Rightarrow_{apply}$	$\langle t_1, e, \mathtt{CONT1}(v_0, k)\rangle$
$\langle \mathtt{CONT1}([t', e'], k), v_1\rangle$	$\Rightarrow_{apply}$	$\langle t', v_1 :: e', k\rangle$
$\langle \mathtt{CONT0}, v\rangle$	$\Rightarrow_{final}$	v

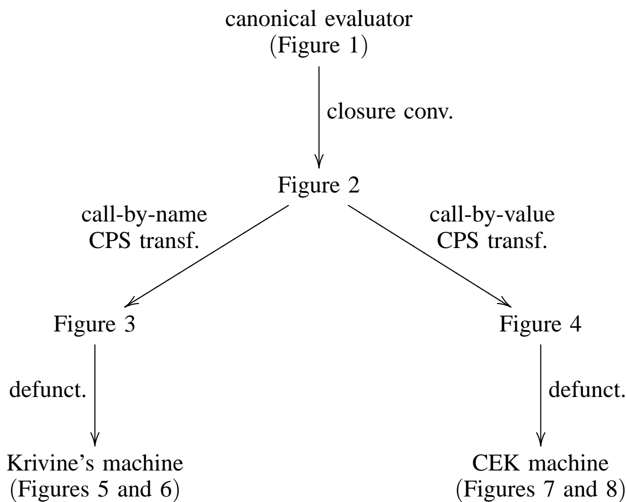
The abstract machine consists of two mutually recursive transition functions. The first transition function operates on triples consisting of a term, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and an expressible value.

Each line in the table matches a clause in Figure 7.

Figure 8. The CEK machine

- Figure 3 displays the call-by-name CPS counterpart of the evaluator of Figure 2.
- Figure 4 displays the call-by-value CPS counterpart of the evaluator of Figure 2.
- Figure 5 displays the defunctionalized version of the evaluator of Figure 3, with the corresponding apply function inlined. Merging the domains of expressible values and of denotable values into one (recursive) domain of thunks pairing terms and environments, and representing the data type `cont` as a list yields the transition function of Krivine's machine (Figure 6).
- Figure 7 displays the defunctionalized version of the evaluator of Figure 4. It corresponds to the transition function of the CEK machine (Figure 8).

Therefore, if the CPS transformation is call-by-name, the resulting transition function is that of Krivine's abstract machine, and if the CPS transformation is call-by-value, the resulting transition function is that of the CEK machine:



Reynolds's point was that in general the evaluation order of the defining language, in a definitional interpreter, determines the evaluation order of the defined language if the definitional interpreter is in direct style (and does not use thunks). The author and his students have recently shown that a call-by-name interpreter leads one to Krivine's abstract machine and that a call-by-value interpreter leads one to the CEK machine [2]. It is a further (and new) consequence of the embodiment of evaluation order in a CPS transformation [41] that Krivine's abstract machine and the CEK machine can in fact be derived from the *same* canonical evaluator. In particular, other CPS transformations would lead to other abstract machines.

Krivine's abstract machine has been discovered, the CEK machine has been invented, and each of them has been celebrated independently and on its own right. Yet, as shown here, they are two sides of the same coin.

4 Consequences

We review further consequences of the connection between evaluation contexts, continuations, and the rest of the computation.

4.1 Designing syntactic theories and abstract machines

Beside making it simple to connect one-step reducers and evaluators, the interpretation of evaluation contexts with a plug function or with a refocus function has direct consequences for designing syntactic theories and abstract machines:

- For programming languages where one can write a one-step reducer using recursive descent, one can mechanically construct the grammar of evaluation contexts and the corresponding plug function, and rest assured that the unique-decomposition lemma holds [70].

  Furthermore, given such a one-step reduction machinery, one can mechanically construct the corresponding abstract machine [22].

- For programming languages where one can write an evaluator using recursive descent, one can mechanically construct the grammar of evaluation contexts and the corresponding refocus function. The result is the transition function of an abstract machine.

  Conversely, one can see abstract machines such as Krivine's machine and the CEK machine as defunctionalized continuation-passing interpreters.

The two points above are not just an academic observation—they have concrete consequences in that they have made it possible for the author and his students to uniformly transform a given evaluator into an abstract machine that was independently invented or discovered, to uniformly exhibit the evaluator underlying a given abstract machine, and to design new evaluators, new abstract machines, and new virtual machines [1, 2]. Beside Krivine's machine and the CEK machine, examples include Landin's SECD machine, Hannan and Miller's CLS machine, Curien et al.'s Categorical Abstract Machine, Schmidt's VEC machine, and Leroy's Zinc machine as well as abstract machines for non-strict functional languages [3], logic-programming languages [11], functional languages with computational effects, including the security technique of stack inspection [4], imperative languages, and object-oriented languages. In clear contrast, such evaluators and machines are usually considered independently and on a case-by-case basis. And when abstract machines are derived, it is the medium (i.e., the derivation) rather than the result that tends to be the message [66, 67].

In particular, starting from a monad-based evaluator for the lambda-calculus, we can pick an arbitrary monad and mechanically construct an evaluator, an abstract machine, and a syntactic theory for the corresponding computational effect. In striking contrast, abstract machines and syntactic theories for computational effects have been designed in isolation and reported as such in the literature.

On the other hand, syntactic theories have also been successfully used in situations where the unique-decomposition lemma does not hold, e.g., Concurrent ML [55]. Such situations would require first-class continuations, which are out of scope here.

4.2 Normalization

Another application of the the insight presented in Section 3 and of the derivation reported at PPDP 2003 [2] concerns (not necessarily type-directed) normalization functions as encountered in the area of normalization by evaluation [9, 16, 25]. The author and his students have derived abstract machines as well as virtual machines for normalization [1]. Specifically, we have shown that a call-by-name normalization function yields a machine that generalizes Krivine's machine, and that a call-by-value normalization function yields a machine that generalizes the CEK machine. In the light of Section 3, though, the author now realizes that these two machines are in fact derived from the *same* normalization function.

In noticeable contrast, existing machines for normalization have been designed in isolation rather than by derivation [15, 36].

4.3 Delimited continuations

In CPS, all calls are tail calls. Yet in some situations, it is very convenient to re-initialize a continuation and to mix CPS with non-tail calls. In a program that re-initializes continuations and where not all calls are tail calls, a continuation no longer represents the rest of the computation. Instead, it is delimited by the re-initialization. Capturing such a continuation yields a first-class continuation that returns to its point of activation. Such first-class continuations can be composed. (In contrast, first-class continuations obtained by `call/cc`-like control operators do not return to their point of activation and therefore they cannot be composed.)

Fifteen years ago, Felleisen introduced an operator to delimit control (a "prompt") together with other operators to abstract delimited control [27]. These control operators were specified using a representation of control as a list of activation records. Delimiting control amounted to putting a mark on this list, abstracting delimited control amounted to making a copy of the list up to the closest mark, and activating a delimited continuation amounted to concatenating the copied list to the current list of activation records [30]. Felleisen's work triggered a series of alternative control operators, all based on representing control as a list of activation records interspersed with control marks [38, 42, 43, 48, 54, 59, 60].

To the author, Felleisen's operator for delimiting control fitted precisely a pervasive pattern of functional programming with layered continuations, together with another control operator, `shift` [20]. Consequently, the two control operators to delimit and to abstract control enjoy a number of applications—in fact, they correspond to computational monads [31]—and they are still the topic of study today [35, 44]. Furthermore, they generalize directly to the CPS hierarchy [10, 19, 23], which also corresponds to layered monads [32].

These two lines of work have been opposed because one represents control as a list of activation records, as in an initial algebra, and the other as a continuation function, as in a final algebra [30]. This opposition continues today when control is only considered as a list of activation records, fit for arbitrary surgery.[1] The two representations, however, could be synergized, e.g., by seeing the former as a defunctionalized version of the latter and by identifying when the latter is not a functional version of the former. For example, Felleisen's $\mathcal{F}^+$ control operator appears to have no CPS counterpart [20, Section 5.3].

Another advantage of characterizing delimited continuations using repeated CPS transformations is that, through the derivation outlined in Section 3 (closure conversion, CPS transformations (note the plural), and defunctionalization), one obtains abstract machines for delimited control [17]. In these machines, delimited control is represented through a series of control stacks, one for each layered continuation.[2]

4.4 Landin's SECD machine

Imagine an environment-based, call-by-value evaluator for the λ-calculus with a callee-save strategy, that furthermore delimits control when evaluating the body of a λ-abstraction. This evaluator operates on the same representation of λ-terms as in Section 3.

```
datatype value = FUN of value -> value

(*  eval : term * value list -> value * value list  *)
fun eval (IND n, e)
    = (List.nth (e, n), e)
  | eval (ABS t, e)
    = (FUN (fn v => reset (fn () => #1 (eval (t, v :: e)))),
       e)
  | eval (APP (t0, t1), e)
    = let val (v1, e) = eval (t1, e)
          val (v0, e) = eval (t0, e)
      in apply (v0, v1, e)
      end
(*  apply : value * value * value list ->
            value * value list  *)
and apply (FUN f, v, e)
    = (f v, e)

(*  evaluate : term -> value  *)
fun evaluate t
    = reset (fn () => #1 (eval (t, nil)))
```

From this evaluator, one can reconstruct Landin's SECD machine as follows:

1. closure conversion of the function space in the domain of values;
   ```
   datatype value = FUN of term * E
   withtype E = value list
   ```
2. introduction of a data stack to hold the intermediate results of `eval`;
   ```
   eval : term * S * E -> S * E
   withtype S = value list
        and E = value list
   ```

[1] The danger of this surgery is that it is so plausible. For example, in the first implementation of Lisp, it was sweepingly plausible to push the bindings of the formals and the actuals on the stack at call time, and to pop them off at return time. The result was dynamic scope.

[2] The author wishes to emphasize this point with an anecdote about Gasbichler and Sperber's implementation of delimited continuations in Scheme 48 [35]. In the course of their work, Gasbichler and Sperber consulted each of the authors of control operators for delimited control, to make sure that their implementation of each delimited-control operator was accurate. This consultation apparently took some time to stabilize. In sharp contrast, it reduced to one e-mail reply from the author, with the guideline of checking that the CPS counterpart of the implementation matches the CPS specification of shift and reset. The next time the author heard of Gasbichler and Sperber's work, it was in the list of accepted papers at ICFP 2002.

3. CPS transformation;

```
eval : term * S * E * C -> value
withtype S = value list
     and E = value list
     and C = S * E -> value
```

4. second CPS transformation, to get rid of the non-tail call due to the presence of `reset`;

```
eval : term * S * E * C * D -> 'a
withtype S = value list
     and E = value list
     and C = S * E * D -> 'a
     and D = value -> 'a
```

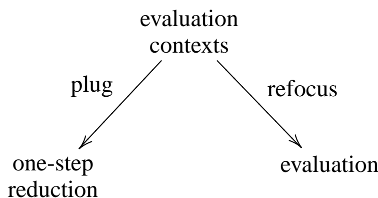
5. defunctionalization of the two layered continuations;
6. fusion of the resulting mutually recursive functions into one.

The SECD machine is a transition function operating on a four-component state: a stack register, an environment register, a control register, and a dump register [46]. The stack register holds the data stack introduced above; the environment register holds the environment threaded in the evaluator above; the control register holds the first continuation in defunctionalized form; and the dump register holds the second continuation in defunctionalized form. We therefore claim that the denotational essence of the SECD machine is this evaluator, with its callee-save strategy for the environment and its control delimiter. The rest—stack register, control register, and dump register—are mere operational artifacts.

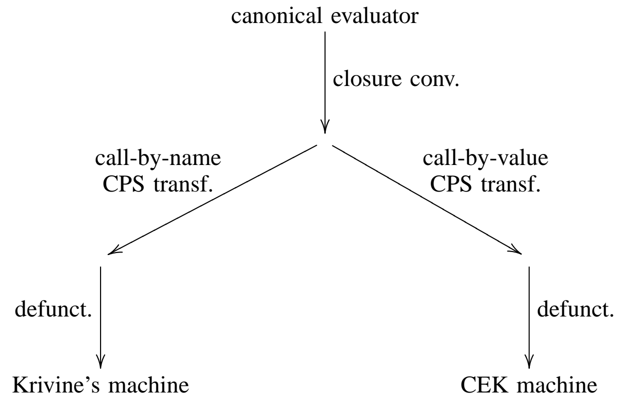
This derivation is documented in a BRICS technical report [18]. It is based on the insight of Section 2 and at the origin of the derivation of Section 3. It solves a long-standing open problem about the particular architecture of the SECD machine, which had never been fully explained—though many variations and simplifications exist. These variations and simplifications (as well as arbitrary new ones) can be obtained by tuning this evaluator and then transforming it into an abstract machine. For example, omitting the control delimiter (which operationally is unused) yields an SEC machine.

5 Conclusion and current work

We have reconciled the notion of continuations as evaluation contexts with the notion of continuations as representations of the rest of the computation. To this end, we have factored the continuation of a single-step reducer and the continuation of an evaluator as the same evaluation contexts with two different interpretations:



As a consequence of this factorization, we have shown that the two best-known abstract machines for the λ-calculus can be derived from the same canonical evaluator for the λ-calculus:



This derivation provides a constructive corollary of Reynolds's historical warning about the evaluation order of defining languages [56] and it scales to normalization functions and abstract machines for normalization. It is an instance of a functional correspondence that lets one reconstruct known abstract machines, construct new ones, e.g., with monadic computational effects, systematically equip them with stack inspection [14], and mechanically construct the corresponding syntactic theories.

In this article, we have considered continuations in the operational setting of reduction, evaluation, and normalization. They are, however, ubiquitous in many other areas, such as semantics [64] and logic [37] as well as in operating-systems services [24, 69].

Acknowledgments:

The author is grateful to Lasse Reichstein Nielsen for our joint study of defunctionalization, which led us to refocusing and the mechanical construction of abstract machines from one-step reduction functions, a study now jointly continued with Małgorzata Biernacka. Thanks are also due to Mads Sig Ager, Dariusz Biernacki, and Jan Midtgaard for our joint study of the correspondence between evaluation functions, abstract machines, and virtual machines, and to Hayo Thielecke for the opportunity to present this work at CW'04.

This article has benefited from Julia Lawall's comprehensive as well as timely comments.

This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`).

6 References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and

lazy abstract machines. Technical Report BRICS RS-03-24, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Accepted for publication in Information Processing Letters.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Technical Report BRICS RS-03-35, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2003.

[5] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[6] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM Press.

[7] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, Sendai, Japan, October 2001. Springer-Verlag.

[8] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, June 1997. ACM Press.

[9] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[10] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004. To appear.

[11] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-03-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Presented at the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).

[12] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[13] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Smolka [61], pages 56–71.

[14] John Clements and Matthias Felleisen. A tail-recursive semantics for stack inspections. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, number 2618 in Lecture Notes in Computer Science, pages 22–37, Warsaw, Poland, April 2003. Springer-Verlag.

[15] Pierre Crégut. An abstract machine for lambda-terms normalization. In Wand [68], pages 333–340.

[16] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[17] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Smolka [61], pages 88–103.

[18] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.

[19] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [68], pages 151–160.

[20] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[21] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.

[22] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. Technical Report BRICS RS-02-04, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[23] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[24] Richard Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 122–136, Pacific Grove, California, October 1991.

[25] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[26] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[27] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

[28] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes.

http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html, 1989-2003.

[29] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[30] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988. ACM Press.

[31] Andrzej Filinski. Representing monads. In Boehm [12], pages 446–457.

[32] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[33] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[34] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[35] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Peyton Jones [50], pages 271–282.

[36] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Peyton Jones [50], pages 235–246.

[37] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[38] Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, California, June 1995. ACM Press.

[39] Chris Hankin. *Lambda Calculi, a guide for computer scientists*, volume 1 of *Graduate Texts in Computer Science*. Oxford University Press, 1994.

[40] Robert Harper. Programming languages: Theory and practice. Working Draft. http://www.cs.cmu.edu/~rwh/plbook/, Spring Semester, 2002.

[41] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [12], pages 458–471.

[42] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, SIGPLAN Notices, Vol. 25, No. 3, pages 128–136, Seattle, Washington, March 1990. ACM Press.

[43] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[44] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

[45] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available online at http://www.logique.jussieu.fr/~krivine, 1985.

[46] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[47] John Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[48] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.

[49] Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000.

[50] Simon Peyton Jones, editor. *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[51] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[52] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[53] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, 1996.

[54] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991. ACM Press.

[55] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[56] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[57] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[58] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[59] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–

99, January 1990.

[60] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [68], pages 161–175.

[61] Gert Smolka, editor. *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, Berlin, Germany, March 2000. Springer-Verlag.

[62] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[63] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.

[64] Hayo Thielecke. From control effects to typed continuation passing. In Greg Morrisett, editor, *Proceedings of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 38, No. 1, pages 139–149, New Orleans, Louisiana, January 2003. ACM Press.

[65] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[66] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.

[67] Mitchell Wand. A semantic prototyping system. In Susan L. Graham, editor, *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, pages 213–221, Montréal, Canada, June 1984. ACM Press.

[68] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[69] Mitchell Wand. Continuation-based multiprocessing. *Higher-Order and Symbolic Computation*, 12(3):285–299, 1999. Reprinted from the proceedings of the 1980 Lisp Conference, with a foreword.

[70] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.

Semantics & Pragmatics Volume 1, Article 1: 1–42, 2008
doi: 10.3765/sp.1.1

Donkey anaphora is in-scope binding[*]

Chris Barker
New York University

Chung-chieh Shan
Rutgers University

Received 2008-01-06 / First Decision 2008-02-29 / Revised 2008-03-23 / Second Decision 2008-03-25 / Revised 2008-03-27 / Accepted 2008-03-27 / Published 2008-06-09

Abstract We propose that the antecedent of a donkey pronoun takes scope over and binds the donkey pronoun, just like any other quantificational antecedent would bind a pronoun. We flesh out this idea in a grammar that compositionally derives the truth conditions of donkey sentences containing conditionals and relative clauses, including those involving modals and proportional quantifiers. For example, an indefinite in the antecedent of a conditional can bind a donkey pronoun in the consequent by taking scope over the entire conditional. Our grammar manages continuations using three independently motivated type-shifters, Lift, Lower, and Bind. Empirical support comes from donkey weak crossover (**He beats it if a farmer owns a donkey*): in our system, a quantificational binder need not c-command a pronoun that it binds, but must be evaluated before it, so that donkey weak crossover is just a special case of weak crossover. We compare our approach to situation-based E-type pronoun analyses, as well as to dynamic accounts such as Dynamic Predicate Logic. A new 'tower' notation makes derivations considerably easier to follow and manipulate than some previous grammars based on continuations.

Keywords: donkey anaphora, continuations, E-type pronoun, type-shifting, scope, quantification, binding, dynamic semantics, weak crossover, donkey pronoun, variable-free, direct compositionality, D-type pronoun, conditionals, situation semantics, c-command, dynamic predicate logic, donkey weak crossover

1 Introduction

A donkey pronoun is a pronoun that lies outside the antecedent of a conditional (or outside the restrictor of a quantifier), yet covaries with some quantificational element

* Thanks to substantial input from Anna Chernilovskaya, Brady Clark, Paul Elbourne, Makoto Kanazawa, Chris Kennedy, Thomas Leu, Floris Roelofsen, Daniel Rothschild, Anna Szabolcsi, Eytan Zweig, and three anonymous referees. Thanks also to Anna Szabolcsi's Spring 2007 seminar, Chris Barker's Spring 2008 seminar, and the Graduate Workshop in Semantics and Philosophy of Language at the University of Chicago.

©2008 Chris Barker and Chung-chieh Shan
This is an open-access article distributed under the terms of a Creative Commons Non-Commercial License (creativecommons.org/licenses/by-nc/3.0).

inside it, usually an indefinite.

(1) a. If a farmer owns a donkey, he beats it.
b. Every farmer who owns a donkey beats it.

Evans 1977 made it standard to assume that the indefinite *a donkey* cannot take scope over the pronoun *it* in (1), and therefore cannot bind it, at least not in the ordinary sense of binding. To the contrary, we claim that the relationship between *a donkey* and *it* in (1) *seems* like binding because it *is* just binding. More specifically, we argue that the indefinites in (1) do take scope over their donkey pronouns, and bind them in the ordinary way, just as a quantifier such as *everyone* takes scope over and binds the pronoun in the bound reading of *Everyone*$_i$ *thinks he*$_i$ *is intelligent*.

As far as we know, no one has ever advocated an in-scope binding analysis of donkey anaphora. It turns out that the right theory of scope and binding makes an in-scope binding analysis not only feasible but straightforward.

1.1 Why not?

One reason people discounted the possibility of in-scope binding in examples like those in (1) is a tradition going back at least to Evans 1977 and May 1977 that says that the scope of all quantifiers is clause bounded.

(2) a. *[Everyone$_i$ arrived] and [she$_i$ spoke].
b. A woman$_i$ arrived and she$_i$ spoke.

If *everyone* can only take scope over the first clausal conjunct in (2a), that explains why it cannot bind the pronoun in the second. But it has been known at least since Farkas 1981 (see Szabolcsi 2007 for a fuller picture) that the scope options for indefinites are strikingly different from those of *every* and certain other quantifiers. And in fact when *everyone* in (2a) is replaced with an indefinite, as in (2b), covariation becomes possible. We will assume, along with many others, that indefinites can take scope wider than their minimal clause, though unlike most, we do not provide any special mechanism like choice functions (e.g., Reinhart 1997) or singleton restricted domains (Schwarzschild 2002).

A second common reason to reject a binding relationship in (1) is the widespread belief that quantificational binding requires c-command.

(3) a. [Everyone$_i$'s mother] loves him$_i$.
b. [Someone from every city$_i$] hates it$_i$.

The universal quantifiers in (3) do not c-command the pronouns that they seem to bind. Büring (2004) concludes that these pronouns, too, are cases of donkey

anaphora, and analyzes them using situations. Shan & Barker (2006) claim that what examples like (3) show is that c-command simply isn't required for quantificational binding. Without concentrating on discussing and defending this claim, we use it in this paper to treat a wide range of examples to good effect.

So we'll assume that the indefinites in (1) can take scope over their respective donkey pronouns and bind them. However, we do not get the desired truth conditions if we give the indefinites in (1) scope over the entire sentence.[1]

(4) a. If a donkey eats, it sleeps.
    b. $\exists d.\, (\mathbf{donkey}\, d) \wedge \big((\mathbf{eats}\, d) \rightarrow (\mathbf{sleeps}\, d)\big)$

If the indefinite takes scope over the entire sentence, we get the truth conditions for (4a) given in (4b). But as Evans (1980: 342) points out, the most natural reading of (4a) is not that there is some donkey that sleeps when it eats, but rather that when *any* donkey eats, that donkey sleeps. Evans concludes that the indefinite must take scope inside the antecedent clause, leaving the pronoun unbound.

Evans' conclusion is hasty. We should conclude only that the indefinite must take scope under the scope of the *if*. Since the *if* takes scope over the entire conditional, it is feasible for the indefinite to also take scope over the entire conditional.

1.2 Sketch of the account

For a rough idea of how the fragment below achieves this result, consider that many treatments of donkey anaphora analyze the conditional as introducing some generic or universal quantification, giving rise to paraphrases such as "For all donkeys d, (eats d) $\rightarrow$ (sleeps d)". (We will discuss more sophisticated modal treatments of the conditional in section 3.3.) Instead of the arrow, as in A $\rightarrow$ B, we use the logically equivalent expression $\neg(\mathrm{A} \wedge \neg\mathrm{B})$. As long as the outer negation takes wide scope, we have (5a) as the truth conditions for (4a).

(5) a. $\neg\exists d.\, \big((\mathbf{donkey}\, d) \wedge (\mathbf{eats}\, d) \wedge \neg(\mathbf{sleeps}\, d)\big)$
    b. $\forall d.\, \neg\big((\mathbf{donkey}\, d) \wedge (\mathbf{eats}\, d) \wedge \neg(\mathbf{sleeps}\, d)\big)$

The indefinite interacts with the negation to give the impression of universal force, since (5a) is logically equivalent to (5b).

On the standard treatment, then, *if* is quantificational, but does not participate in scope ambiguities in the same way as other quantificational operators. On our proposal, the scope of *if* interacts with other scope-taking elements in the normal way. Section 3 shows how to arrive at the analysis in (5) compositionally.

1 We adopt the standard convention that a dot following a binding operator indicates that the scope of the operator extends as far to the right as possible. Thus, $\exists d.\, \phi \wedge \psi$ is shorthand for $\exists d(\phi \wedge \psi)$.

A similar challenge arises in the case of donkey anaphora from relative clauses.

(6) a. Most men who own a car wash it on Sundays.
    b. Every man who owns a donkey beats it.

Evans (1977: 117) provides an influential assessment:

> If the sentence is to express the intended restrictions upon the major quantifier—that of being a car- or donkey-owner—it would appear that the second quantifier must be given a scope which does not extend beyond the relative clause, and this rules out a bound variable interpretation of the later pronouns.

Once again, appearances are deceiving: the scope of the indefinite need not be limited to the relative clause, even on the intended reading. Section 4 shows how an independently motivated lexical entry for *every* gives rise to the following truth conditions.

(7) a. Every farmer who owns a donkey beats it.
    b. $\neg\exists x\exists y.\, \mathbf{donkey}\, y \wedge \big((\mathbf{farmer}\, x \wedge \mathbf{owns}\, y\, x) \wedge \neg(\mathbf{beats}\, y\, x)\big)$

In section 4, we also generalize this analysis to strong and weak readings of the indefinite with arbitrary quantifiers.

In sum, as long as an indefinite can take scope outside its minimal clause (possibly a relative clause), donkey anaphora falls out from independently motivated compositional semantics.

1.3 Supporting evidence: donkey weak crossover

We cannot respond to all of the vast literature on donkey anaphora in the compass of a single paper. Nevertheless, we should and will evaluate our proposal against some prominent alternatives. Elbourne (2005) presents a recent survey of approaches to donkey anaphora. His own proposal is carefully motivated and empirically robust, and provides stiff competition for any new theory of donkey anaphora. Elbourne analyzes donkey pronouns as covert definite descriptions ("D-type" pronouns) that seek their referents within severely restricted situations.

One challenge for the D-type approach is the celebrated bishop problem, based on examples such as (8).

(8) If a bishop$_i$ meets a bishop$_j$, he$_i$ blesses him$_j$.

As shown below in section 3.1, bishop sentences are perfectly straightforward on a binding account: the first indefinite binds one pronoun, and the second indefinite binds the other pronoun.

Elbourne argues specifically against dynamic treatments and variable-free treatments. Since our system can be seen as both dynamic and variable-free, we will consider several of Elbourne's arguments. One telling argument that Elbourne advances against Groenendijk & Stokhof's (1989; 1991) dynamic treatment (to be discussed in section 7) involves disjunctive antecedents.

(9) If a farmer owns a donkey or a goat, he beats it.

Given the independently motivated theory of generalized disjunction in Barker 2002, we show in section 5 that such examples fall out without further stipulation in our system. Thus Elbourne's arguments against dynamic treatments of donkey anaphora apply not in general but only to specific dynamic theories.

One appeal of dynamic semantics is that at least quantificational anaphora is clearly sensitive to order.

(10) a. A woman$_i$ arrived and she$_i$ spoke.
b. *She$_i$ arrived and a woman$_i$ spoke.

Only the example (10a), in which the indefinite precedes the pronoun, allows binding.

An in-scope binding analysis of donkey anaphora, then, predicts similar order sensitivity. It is well known (e.g., Büring 2004) that donkey anaphora out of a DP exhibits crossover effects.

(11) a. Most women who have a son$_i$ love his$_i$ father.
b. *His$_i$ father loves most women who have a son$_i$.

Crossover occurs when a quantificational expression (*most women who have a son*) takes scope over a pronoun that linearly precedes it. It is called crossover because, on a movement approach such as Quantifier Raising, the quantificational DP crosses over the position of the pronoun. As the contrast in (11) shows, donkey anaphora requires the indefinite to precede the covarying pronoun, even though the quantifier *most* takes scope over the entire sentence in both (11a) and (11b).

Büring derives the contrast in (11) from his assumption that a DP that contains a donkey antecedent must c-command the donkey pronoun. Without invoking c-command, we explain donkey weak crossover just as we explain weak crossover.

What is less well-known is that donkey anaphora in a conditional is sensitive to order just as donkey anaphora out of a DP is.

(12) a. If a farmer owns a donkey, he beats it. (= (1a))
b.*?If he owns it, a farmer beats a donkey.

The D-type account correctly predicts that anaphora from the antecedent into the consequent is good, as in the standard sentence repeated in (12a), and that similar

anaphora from the consequent into the antecedent is significantly more difficult, as in (12b).

Although the contrast in (12) appears to be reliable across speakers, some judge (12b) as not so bad in absolute terms. Indeed, some cases of donkey cataphora are unexpectedly good,[2] as in this example from Chierchia (1995: 129):

(13) If it is overcooked, a hamburger usually doesn't taste good.

However, if the donkey antecedent is not in subject position, or if the sentence is episodic (Reinhart 1983: 115–116), acceptability degrades significantly:[3]

(14) a. *If it is overcooked, John doesn't like a hamburger.
b. *If John overcooked it, a hamburger tasted bad.

As predicted by our analysis, if the order of the *if* clause and the main clause are reversed, the donkey anaphora becomes effortless.

In any case, if we return the *if* clause to its canonical position as a subordinating conjunction, as in (15), judgments are considerably more robust.

(15) a. A farmer beats a donkey if he owns it.
b. *He beats it if a farmer owns a donkey.

The anaphora from the consequent into the antecedent in (15a) is good, but the anaphora from the antecedent into the consequent in (15b) is bad. An account based on situations at best fails to predict this contrast.

1.4 Dynamic semantics?

Before getting to the analysis, a brief note on our conception of dynamic semantics. We consider our analysis dynamic, but not quite in the traditional sense of, say, Heim 1982 or Groenendijk & Stokhof 1991. There, sentence meanings are conceived as context update functions: functions that map an initial information state to an updated information state. On our view, the essence of a dynamic analysis is a principled distinction between the side effects of an expression and its main semantic contribution (Shan 2005). Here, the main contribution is to local argument structure, and the side effects are potentially long-distance semantic relationships, including

2 That is, unexpected given our analysis. We claim that c-command is not necessary for quantificational anaphora, but c-command or something like it could still facilitate some types of cataphora, as in Reinhart's (1983) *Near him, John saw a snake*.

3 In order to guarantee that we have cataphora, it is important to consider the examples in (14) in a context in which hamburgers have not been given as a pre-established topic. In (14a), one way to achieve this is by putting the default nuclear stress pitch accent on *HAMburger*.

scope-taking and binding. Our denotations are the ordinary ones found in any extensional semantics: individuals, truth values, and functions built from them. There is no need to assume that expressions directly manipulate the pragmatic context, whether it is a set of worlds, a set of assignment functions, or another kind of information state. Thus on our analysis, expressions can be viewed as denoting updates, but what gets updated is the semantic context, not the utterance context.

See section 7 for a more detailed comparison with other dynamic treatments of donkey anaphora, including Groenendijk & Stokhof's (1989) Dynamic Montague Grammar and Shan's (2001) and de Groote's (2006) variable-free analyses based on continuations.

2 Fragment

Compositional truth conditions are the crux of our claim, so we will present our analysis in detail as a concrete fragment. The fragment is directly compositional and variable-free in the sense of Jacobson 1999, although that plays no special role in the explanations below. More crucial is our use of continuations, heralded by the fact that we give a categorial grammar with two flavors of slashes: not just $\backslash$ and $/$, which encode syntactic adjacency on the left and on the right respectively, but also $\backslash\!\!\backslash$ and $/\!\!/$, which, as we will see, govern scope-taking where an in-situ quantifier is pronounced and where it takes scope.

The fragment here is identical to a part of the system presented in Shan & Barker 2006 (see the appendix below for details of the correspondence). However, the presentation here is different and, we hope, simpler. There are three sources of simplicity. First, we need only present the mechanisms for scope-taking and binding, not *wh*-movement. Second, although the system relies heavily on continuations, we do not pause here to motivate them or discuss their theory; such discussions can be found in Barker 2002, Shan 2005, and Shan & Barker 2006. Third, in this paper we adopt a notation that makes the derivations far easier to understand compared with the derivations in Shan & Barker 2006.

Our new notation consists of a syntactic part and a semantic part. In syntactic categories, we write $B/\!\!/(A\backslash\!\!\backslash C)$ vertically as $\dfrac{B\,|\,C}{A}$ and omit the double slashes. Here A,B,C can be any categories. For example, we write the category $\mathrm{S}/\!\!/(\mathrm{DP}\backslash\!\!\backslash\mathrm{S})$ as $\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}$. In semantic values, we write $\lambda\kappa.\, f[\kappa(x)]$ vertically as $\dfrac{f[\;]}{x}$ and omit κ. Here x can be any expression, and $f[\;]$ can be any expression with a hole $[\;]$. Free variables in x can be bound by binders in $f[\;]$. For example, we write the value $\lambda\kappa.\, \neg\exists x.\, \kappa(\mathbf{mother}\, x)$ as $\dfrac{\neg\exists x.\,[\;]}{\mathbf{mother}\, x}$, in which the variable x in $\mathbf{mother}\, x$ is bound

by $\exists x$. We call the original style 'linear' notation, and the new style 'tower' notation.

This tower notation will become clearer as we use it below. We first derive quantificational sentences with linear (surface) scope, then turn to inverse scope and binding.

2.1 The tower notation: taking scope

We'll stack information about each expression like this:

$$(16)\qquad \begin{array}{cc} \mathrm{DP} & \text{syntactic category} \\ \textit{John} & \text{expression} \\ \mathbf{j} & \text{semantic value} \end{array}$$

In the simplest case, syntactic combination proceeds as in standard combinatory categorial grammar. For example, we derive *John left* as follows.

$$(17)\qquad \left(\begin{array}{cc} \mathrm{DP} & \mathrm{DP}\backslash\mathrm{S} \\ \textit{John} & \textit{left} \\ \mathbf{j} & \mathbf{left} \end{array}\right) = \begin{array}{c} \mathrm{S} \\ \textit{John left} \\ \mathbf{left}\,\mathbf{j} \end{array}$$

As usual, the category under the slash (here DP) cancels with the category of the argument expression, and the semantics is function application.

Quantificational expressions have an extra layer on top of their syntactic category and on top of their semantic value. For example, below is a simplistic lexical entry for *everyone*.

$$(18)\qquad \begin{array}{c} \dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}} \\ \textit{everyone} \\ \dfrac{\forall y.\,[\;]}{y} \end{array}$$

The syntactic category can be read counterclockwise, starting below the horizontal line:

$$(19)\qquad \frac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}} \quad \text{means} \quad \frac{\ldots\text{to form an S.} \;\Big|\; \text{and takes scope at an S}\ldots}{\text{The expression functions in local syntax as a DP,}}$$

Saying that a DP like *everyone* "takes scope at an S" means that its nuclear scope is formed from an enclosing expression of category S.

We can derive a simple quantificational sentence as follows.

(20)
$$\left(\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}\\ \textit{everyone}\\ \dfrac{\forall y.\,[\,]}{y}\end{array}\quad\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}\backslash\mathrm{S}}\\ \textit{left}\\ \dfrac{[\,]}{\textbf{left}}\end{array}\right) = \begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{S}}\\ \textit{everyone left}\\ \dfrac{\forall y.\,[\,]}{\textbf{left}\,y}\end{array}$$

In the general case, we have the following modes of combination.

(21)
$$\left(\begin{array}{c}\dfrac{C\,|\,\boldsymbol{D}}{A/\boldsymbol{B}}\\ \textit{left-exp}\\ \dfrac{g[\,]}{f}\end{array}\quad\begin{array}{c}\dfrac{\boldsymbol{D}\,|\,E}{\boldsymbol{B}}\\ \textit{right-exp}\\ \dfrac{h[\,]}{x}\end{array}\right) = \begin{array}{c}\dfrac{C\,|\,E}{A}\\ \textit{left-exp right-exp}\\ \dfrac{g[h[\,]]}{f(x)}\end{array}$$
$$\left(\begin{array}{c}\dfrac{C\,|\,\boldsymbol{D}}{\boldsymbol{B}}\\ \textit{left-exp}\\ \dfrac{g[\,]}{x}\end{array}\quad\begin{array}{c}\dfrac{\boldsymbol{D}\,|\,E}{\boldsymbol{B}\backslash A}\\ \textit{right-exp}\\ \dfrac{h[\,]}{f}\end{array}\right) = \begin{array}{c}\dfrac{C\,|\,E}{A}\\ \textit{left-exp right-exp}\\ \dfrac{g[h[\,]]}{f(x)}\end{array}$$

Below the horizontal lines, combination proceeds simply as in combinatory categorial grammar: in the syntax, B combines with A/B or $B\backslash A$ to form A; in the semantics, x combines with f to form $f(x)$.

Above the lines is where the combination machinery for continuations kicks in. The syntax combines the two pairs of categories by a kind of cancellation: the D on the left cancels with the D on the right. The semantics combines the two expressions with holes by a kind of composition: we plug $h[\,]$ to the right into the hole of $g[\,]$ to the left, to form $g[h[\,]]$. The expression with a hole on the left, $g[\,]$, always surrounds the expression with a hole on the right, $h[\,]$, no matter which side supplies the function and which side supplies the argument below the lines. This fact expresses the generalization that the default order of semantic evaluation is left-to-right.

Type-shifter 1 of 3: Lift Comparing the analysis above of *John left* in (17) with that of *Everyone left* in (20) reveals that *left* has been given two distinct values. The first, simpler value is the basic lexical entry, and we derive the more complex value through the standard type-shifter Lift, proposed by Partee & Rooth (1983), Jacobson

(1999), Steedman (2000), and many others.

(22)
$$\begin{array}{c}\mathrm{DP}\backslash\mathrm{S}\\ \textit{left}\\ \textbf{left}\end{array}\quad\begin{array}{c}\text{Lift}\\ \Rightarrow\end{array}\quad\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}\backslash\mathrm{S}}\\ \textit{left}\\ \dfrac{[\,]}{\textbf{left}}\end{array}$$

In general, for any categories A and B and any value x, the following type-shifter is available.

(23)
$$\begin{array}{c}A\\ \textit{expression}\\ x\end{array}\quad\begin{array}{c}\text{Lift}\\ \Rightarrow\end{array}\quad\begin{array}{c}\dfrac{B\,|\,B}{A}\\ \textit{expression}\\ \dfrac{[\,]}{x}\end{array}$$

Syntactically, Lift adds a layer with arbitrary (but matching!) syntactic categories. Semantically, it adds a layer with empty brackets.

Type-shifter 2 of 3: Lower The semantic value given above for *Everyone left* was $\dfrac{\forall x.\,[\,]}{\textbf{left}\,x}$, which corresponds to the syntactic category $\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{S}}$. To derive the syntactic category S and a semantic value with no horizontal line, we introduce the type-shifter Lower. In general, for any category A, any value x, and any semantic expression $f[\,]$ with a hole $[\,]$, the following type-shifter is available.

(24)
$$\begin{array}{c}\dfrac{A\,|\,\mathrm{S}}{\mathrm{S}}\\ \textit{expression}\\ \dfrac{f[\,]}{x}\end{array}\quad\begin{array}{c}\text{Lower}\\ \Rightarrow\end{array}\quad\begin{array}{c}A\\ \textit{expression}\\ f[x]\end{array}$$

Syntactically, Lower cancels an S above the line to the right with an S below the line. Semantically, Lower collapses a two-level meaning into a single level by plugging the value x below the line into the hole $[\,]$ in the expression $f[\,]$ above the line.[4] For

4 As the following examples show, this insertion may capture bound variables. This capture should not cause concern, since the equivalent system in Shan & Barker 2006 is presented entirely without the tower notation, and does not use this variable-capturing device. In linear notation, the semantics of the Lower type-shifter is simply $\lambda F.\, F(\lambda x.x)$. See the appendix for further details.

example, applying Lower to the analysis above for *Everyone left* gives

(25)
$$\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{S}} \\ \textit{everyone left} \\ \dfrac{\forall y.\,[\,]}{\textbf{left}\,y}\end{array} \quad \overset{\text{Lower}}{\Rightarrow} \quad \begin{array}{c}\mathrm{S} \\ \textit{everyone left} \\ \forall y.\,\textbf{left}\,y\end{array}$$

As discussed in section 7, this type-shifter resembles a type-shifting rule proposed by Groenendijk & Stokhof (1989), and is characteristic of continuation-based systems in general (such as in the simulation theorem in Plotkin 1975).

We can now derive a sentence that contains more than one quantifier. Like the intransitive verb *left* above, the transitive verb *loves* is Lifted to match the levels of *someone* and *everyone*.

(26)
$$\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}} \\ \textit{someone} \\ \dfrac{\exists x.\,[\,]}{x}\end{array} \left(\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{(\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP}} \\ \textit{loves} \\ \dfrac{[\,]}{\textbf{loves}}\end{array} \quad \begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}} \\ \textit{everyone} \\ \dfrac{\forall y.\,[\,]}{y}\end{array} \right)$$

$$= \begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{S}} \\ \textit{Someone loves everyone} \\ \dfrac{\exists x.\,\forall y.\,[\,]}{\textbf{loves}\,y\,x}\end{array} \quad \overset{\text{Lower}}{\Rightarrow} \quad \begin{array}{c}\mathrm{S} \\ \textit{Someone loves everyone} \\ \exists x.\,\forall y.\,\textbf{loves}\,y\,x\end{array}$$

Note that this derivation gives linear scope.

The lexical entry for *someone* just used generalizes to an entry for the indefinite determiner *a*.

(27)
$$\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}\Big/\mathrm{N} \\ \mathrm{a} \\ \lambda P.\,\dfrac{\exists x.\,Px \wedge [\,]}{x}\end{array}$$

This generalization shows how a tower can occur as a subpart of a syntactic category or semantic value, because a tower is just shorthand for a category or value. (Section 3.1 shows *a* in action.)

2.2 Multiple layers and inverse scope

Understanding how inverse scope works will turn out to be important later for handling examples involving multiple donkey pronouns.

In order to arrive at inverse scope, we must apply the Lift operator to lower levels of an expression. For example, beginning with the lexical entry for *someone*, there are two distinct ways that we can apply Lift. The first way targets the entire category $\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}$ and the entire meaning $\dfrac{\exists x.\,[\,]}{x}$, and contributes the bold S's and brackets in the result below, on the top level. The existential quantifier ends up on the middle level of the semantic tower.

(28)
$$\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}} \\ \textit{someone} \\ \dfrac{\exists x.\,[\,]}{x}\end{array} \quad \overset{\text{Lift}}{\Rightarrow} \quad \begin{array}{c}\mathbf{S}\,|\,\mathbf{S} \\ \hline \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP} \\ \textit{someone} \\ \mathbf{[\,]} \\ \hline \exists x.\,[\,] \\ \hline x\end{array}$$

The second way targets only the lower-level category DP in the syntax and the lower-level meaning x in the semantics, replacing them with Lifted versions without disturbing the rest of the expression. This way contributes the bold S's and brackets in the result below, on the middle level. This time, crucially, the existential quantifier ends up on the top level of the semantic tower.[5]

(29)
$$\begin{array}{c}\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}} \\ \textit{someone} \\ \dfrac{\exists x.\,[\,]}{x}\end{array} \quad \overset{\text{Lift}}{\Rightarrow} \quad \begin{array}{c}\mathrm{S}\,|\,\mathrm{S} \\ \hline \mathbf{S}\,|\,\mathbf{S} \\ \hline \mathrm{DP} \\ \textit{someone} \\ \exists x.\,[\,] \\ \hline \mathbf{[\,]} \\ \hline x\end{array}$$

Other type-shifters can also target lower levels of an expression. For example,

5 More technically, if $A = \ldots A_0 \ldots$ and $B = \ldots B_0 \ldots$ are two tower categories that differ only in the categories A_0 and B_0 at their bottoms, and some type-shifter T shifts A_0 to B_0, then T also shifts A to B. Analogously, if $x = \ldots x_0 \ldots$ and $y = \ldots y_0 \ldots$ are two tower meanings that differ only in the expressions x_0 and y_0 at their bottoms, and some type-shifter T shifts x_0 to y_0, then T also shifts x to y.

Lower can apply to the bottom two levels of a three-level expression.

(30)
$$\begin{array}{c} \begin{array}{c|c} B & C \\ \hline A & \mathbf{S} \\ \hline \multicolumn{2}{c}{\mathbf{S}} \end{array} \\ \textit{expression} \\ \dfrac{\dfrac{g[\,]}{f[\,]}}{x} \end{array} \quad \overset{\text{Lower}}{\Rightarrow} \quad \begin{array}{c} \begin{array}{c|c} B & C \\ \hline \multicolumn{2}{c}{A} \end{array} \\ \textit{expression} \\ \dfrac{g[\,]}{f[x]} \end{array}$$

Furthermore, binary combination can also target lower levels of a pair of expressions. In other words, binary combination generalizes to multilevel derivations:

- At the bottom level, the syntax cancels a slash while the semantics applies a function. The category of the argument must match the category under the slash (shown as ***B*** below).
- At each level above, the syntax cancels adjacent categories while the semantics composes expressions with holes. The adjacent categories must match at each level (shown as ***D*** and ***G*** below in the case of two levels).

(31)
$$\left(\begin{array}{c} \begin{array}{c|c} F & \boldsymbol{G} \\ \hline C & \boldsymbol{D} \\ \hline \multicolumn{2}{c}{A/\boldsymbol{B}} \end{array} \\ \textit{left-exp} \\ \dfrac{\dfrac{i[\,]}{g[\,]}}{f} \end{array} \quad \begin{array}{c} \begin{array}{c|c} \boldsymbol{G} & H \\ \hline \boldsymbol{D} & E \\ \hline \multicolumn{2}{c}{\boldsymbol{B}} \end{array} \\ \textit{right-exp} \\ \dfrac{\dfrac{j[\,]}{h[\,]}}{x} \end{array}\right) = \begin{array}{c} \begin{array}{c|c} F & H \\ \hline C & E \\ \hline \multicolumn{2}{c}{A} \end{array} \\ \textit{left-exp right-exp} \\ \dfrac{\dfrac{i[j[\,]]}{g[h[\,]]}}{f(x)} \end{array}$$

(32)
$$\left(\begin{array}{c} \begin{array}{c|c} F & \boldsymbol{G} \\ \hline C & \boldsymbol{D} \\ \hline \multicolumn{2}{c}{\boldsymbol{B}} \end{array} \\ \textit{left-exp} \\ \dfrac{\dfrac{i[\,]}{g[\,]}}{x} \end{array} \quad \begin{array}{c} \begin{array}{c|c} \boldsymbol{G} & H \\ \hline \boldsymbol{D} & E \\ \hline \multicolumn{2}{c}{\boldsymbol{B}\backslash A} \end{array} \\ \textit{right-exp} \\ \dfrac{\dfrac{j[\,]}{h[\,]}}{f} \end{array}\right) = \begin{array}{c} \begin{array}{c|c} F & H \\ \hline C & E \\ \hline \multicolumn{2}{c}{A} \end{array} \\ \textit{left-exp right-exp} \\ \dfrac{\dfrac{i[j[\,]]}{g[h[\,]]}}{f(x)} \end{array}$$

As long as *someone* uses the first Lifting method and *everyone* uses the second,

we end up with inverse scope.

(33)
$$\begin{array}{c} \begin{array}{c|c} \mathbf{S} & \mathbf{S} \\ \hline \mathrm{S} & \mathrm{S} \\ \hline \multicolumn{2}{c}{\mathrm{DP}} \end{array} \\ \text{someone} \\ \dfrac{\dfrac{[\,]}{\exists x.\,[\,]}}{x} \end{array} \left(\begin{array}{c} \begin{array}{c|c} \mathrm{S} & \mathrm{S} \\ \hline \mathrm{S} & \mathrm{S} \\ \hline \multicolumn{2}{c}{(\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP}} \end{array} \\ \text{loves} \\ \dfrac{\dfrac{[\,]}{[\,]}}{\textbf{loves}} \end{array} \quad \begin{array}{c} \begin{array}{c|c} \mathrm{S} & \mathrm{S} \\ \hline \mathbf{S} & \mathbf{S} \\ \hline \multicolumn{2}{c}{\mathrm{DP}} \end{array} \\ \text{everyone} \\ \dfrac{\dfrac{\forall y.\,[\,]}{[\,]}}{y} \end{array}\right) = \begin{array}{c} \begin{array}{c|c} \mathrm{S} & \mathrm{S} \\ \hline \mathrm{S} & \mathrm{S} \\ \hline \multicolumn{2}{c}{\mathrm{S}} \end{array} \\ \text{someone loves everyone} \\ \dfrac{\dfrac{\forall y.\,[\,]}{\exists x.\,[\,]}}{\textbf{loves}\,y\,x} \end{array}$$

$$\overset{\text{Lower (twice)}}{\Rightarrow} \quad \begin{array}{c} \mathrm{S} \\ \text{someone loves everyone} \\ \forall y.\,\exists x.\,\textbf{loves}\,y\,x \end{array}$$

To match the additional level produced by Lifting *someone* and *everyone*, we Lift the verb *loves* twice: first using the first Lifting method, then using either Lifting method. To reduce the combined three-level meaning to a single level, we first apply Lower to the bottom two levels, then to the resulting two-level meaning in its entirety.

In general, quantifiers on higher levels outscope lower quantifiers. Within a given level, as demonstrated in section 2.1, quantifiers on the left outscope quantifiers on the right.

2.3 Binding

There are two halves to any binding relationship: the pronoun must create a need to be bound, and the binder must satisfy that need. We accomplish the first half by giving pronouns a lexical entry that announces a functional dependence on an entity-type argument:

(34)
$$\begin{array}{c} \begin{array}{c|c} \mathrm{DP} \rhd B & B \\ \hline \multicolumn{2}{c}{\mathrm{DP}} \end{array} \\ \textit{he} \\ \dfrac{\lambda y.\,[\,]}{y} \end{array}$$

Here B is any category, and the category $\mathrm{DP} \rhd B$ denotes functions from DP to B. For instance, we derive the sentence *He left* below.

(35) $$\left(\begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP} \\ \textit{he} \\ \lambda y.[\,] \\ \hline y \end{array} \quad \begin{array}{c} \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP}\backslash\mathrm{S} \\ \textit{left} \\ [\,] \\ \hline \textbf{left} \end{array}\right) = \begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{S} \\ \textit{He left} \\ \lambda y.[\,] \\ \hline \textbf{left}\, y \end{array} \quad \overset{\text{Lower}}{\Rightarrow} \quad \begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \\ \textit{He left} \\ \lambda y.\, \textbf{left}\, y \end{array}$$

On this view, sentences with free pronouns do not have the same category as sentences without pronouns (e.g., *John left*). This difference captures the fact that a sentence containing a free pronoun does not express a complete thought until the value of the pronoun has been specified, whether by binding or by the pragmatic context. However, the syntactic commonality of the two sentence types is still captured: below the lowest horizontal line, they are both S's, and it is only above the line that their semantic differences are recorded.[6]

Type-shifter 3 of 3: Binding The Bind type-shifter provides the second half of the binding relationship: it allows an arbitrary DP to control the value of a subsequent pronoun. For any categories A and B, any value x, and any semantic expression $f[\,]$ with a hole $[\,]$, the following type-shifter is available.

(36) $$\begin{array}{c} A \,|\, B \\ \hline \mathrm{DP} \\ \textit{expression} \\ f[\,] \\ \hline x \end{array} \quad \overset{\text{Bind}}{\Rightarrow} \quad \begin{array}{c} A \,|\, \mathrm{DP} \rhd B \\ \hline \mathrm{DP} \\ \textit{expression} \\ f([\,]x) \\ \hline x \end{array}$$

Syntactically, Bind annotates the top right syntactic category with "DP $\rhd$", offering to bind a following pronoun. Semantically, Bind copies the value of the DP and feeds it to the pronoun. For instance, applying Bind to *everyone*, we have

(37) $$\begin{array}{c} \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP} \\ \textit{everyone} \\ \forall x.[\,] \\ \hline x \end{array} \quad \overset{\text{Bind}}{\Rightarrow} \quad \begin{array}{c} \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline \mathrm{DP} \\ \textit{everyone} \\ \forall x.([\,]x) \\ \hline x \end{array}$$

Note the extra copy of x in the semantics of the shifted (binding) version.

6 Like Jacobson (e.g., 1999), but unlike Dowty (2007), we assume that pronouns denote identity functions. Like Dowty, but unlike Jacobson, we explicitly recognize that pronouns literally take scope.

This move lets us complete the following derivation. (To simplify the exposition, we treat *his* like *him* and assign *mother* the functional category DP\DP.)

(38) $$\begin{array}{c} \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline \mathrm{DP} \\ \textit{everyone} \\ \forall x.([\,]x) \\ \hline x \end{array} \left(\begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline (\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP} \\ \textit{loves} \\ [\,] \\ \hline \textbf{loves} \end{array} \left(\begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP} \\ \textit{his} \\ \lambda y.[\,] \\ \hline y \end{array} \quad \begin{array}{c} \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP}\backslash\mathrm{DP} \\ \textit{mother} \\ [\,] \\ \hline \textbf{mother} \end{array}\right)\right)$$

$$= \begin{array}{c} \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{S} \\ \textit{Everyone loves his mother} \\ \forall x.((\lambda y.[\,])x) \\ \hline \textbf{loves}\,(\textbf{mother}\, y)\, x \end{array} \quad \overset{\text{Lower}}{\Rightarrow} \quad \begin{array}{c} \mathrm{S} \\ \textit{Everyone loves his mother} \\ \forall x.((\lambda y.\, \textbf{loves}\,(\textbf{mother}\, y)\, x)x) \end{array}$$

This lowered value reduces to $\forall x.\, \textbf{loves}\,(\textbf{mother}\, x)\, x$, as desired for the bound reading. This derivation Lifts the verb *loves* from the category (DP\S)/DP to $\dfrac{\mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S}}{(\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP}}$ rather than $\dfrac{\mathrm{S} \,|\, \mathrm{S}}{(\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP}}$, so as to cancel first against *his mother* on the right and then against *everyone* on the left. Our Lift type-shifter allows this move because its schematic category B can be instantiated as DP $\rhd$ S or any other category, not just the S used to lift *mother* above.

2.4 Binding without c-command; the dynamics of weak crossover

Binding without c-command is crucial to our account of donkey anaphora. Since c-command has no special status in our theory of binding, it is perfectly possible to have binding without c-command:

(39) $$\left(\begin{array}{c} \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline \mathrm{DP} \\ \text{everyone's} \end{array} \quad \begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline \mathrm{DP}\backslash\mathrm{DP} \\ \text{mother} \end{array}\right) \left(\begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline (\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP} \\ \text{loves} \end{array} \quad \begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP} \\ \text{him} \end{array}\right)$$

This derivation yields the final interpretation $\forall y.\, \textbf{loves}\, y\, (\textbf{mother}\, y)$.

In contrast, the order of the binder and the pronoun is crucial.

(40) $$\left(\begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP} \\ \text{his} \end{array} \quad \begin{array}{c} \mathrm{S} \,|\, \mathrm{S} \\ \hline \mathrm{DP}\backslash\mathrm{DP} \\ \text{mother} \end{array}\right) \left(\begin{array}{c} \mathrm{S} \,|\, \mathrm{S} \\ \hline (\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP} \\ \text{loves} \end{array} \quad \begin{array}{c} \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline \mathrm{DP} \\ \text{everyone} \end{array}\right)$$

$$= \begin{array}{c} \mathrm{DP} \rhd \mathrm{S} \,|\, \mathrm{DP} \rhd \mathrm{S} \\ \hline \mathrm{S} \\ \text{his mother loves everyone} \end{array}$$

In this weak crossover configuration, syntactic cancellation goes through without any difficulty up until the point at which we would like to apply Lower. But the schema for triggering the Lower rule isn't met, since lowering requires the top right category to be S, not DP $\triangleright$ S. At best, the pronoun mournfully looks left for an antecedent, while the binder looks right for a pronoun to bind. Thus we correctly predict that there is no complete derivation for sentences in which the pronoun is evaluated before its binder.

3 Donkey anaphora in conditionals

Typical donkey anaphora in conditionals involves an indefinite in the antecedent clause covarying with a pronoun in the consequent.

As discussed in section 1, we need to articulate the denotation of *if* into an in-situ part and a scope-taking part:

(41)
$$\begin{array}{c} \mathrm{S} \mid \mathrm{S} \\ \hline (\mathrm{S}/\mathrm{S})/\mathrm{S} \\ \textit{if} \\ \neg[\,] \\ \hline \lambda p\lambda q.\, p \wedge \neg q \end{array}$$

Crucially, in the semantic value the outer negation is above the line, where it can take scope over the entire conditional. The conjunction and the inner negation are below the line, where they can distinguish the antecedent from the consequent.

(42)
$$\begin{array}{c} \mathrm{S} \mid \mathrm{S} \\ \hline (\mathrm{S}/\mathrm{S})/\mathrm{S} \\ \textit{if} \\ \neg[\,] \\ \hline \lambda p\lambda q.\, p \wedge \neg q \end{array} \left(\begin{array}{c} \mathrm{S} \mid \mathrm{DP} \triangleright \mathrm{S} \\ \hline \mathrm{DP} \\ \textit{someone} \\ \exists y.\,([\,]\,y) \\ \hline y \end{array} \quad \begin{array}{c} \mathrm{DP} \triangleright \mathrm{S} \mid \mathrm{DP} \triangleright \mathrm{S} \\ \hline \mathrm{DP}\backslash\mathrm{S} \\ \textit{knocked} \\ [\,] \\ \hline \textbf{knocked} \end{array} \right) \left(\begin{array}{c} \mathrm{DP} \triangleright \mathrm{S} \mid \mathrm{S} \\ \hline \mathrm{DP} \\ \textit{she} \\ \lambda x.\,[\,] \\ \hline x \end{array} \quad \begin{array}{c} \mathrm{S} \mid \mathrm{S} \\ \hline \mathrm{DP}\backslash\mathrm{S} \\ \textit{left} \\ [\,] \\ \hline \textbf{left} \end{array} \right)$$

The indefinite takes scope over the consequent and binds the pronoun, which accounts for covariance. The outer negation contributed by the *if* takes even wider scope. After combination, we have

(43)
$$\begin{array}{c} \mathrm{S} \mid \mathrm{S} \\ \hline \mathrm{S} \\ \neg\exists y.\,((\lambda x.\,[\,])\,y) \\ \hline (\textbf{knocked}\,y) \wedge \neg(\textbf{left}\,x) \end{array} \overset{\text{Lower}}{\Rightarrow} \begin{array}{c} \mathrm{S} \\ \neg\exists y.\,(\textbf{knocked}\,y) \wedge \neg(\textbf{left}\,y) \end{array}$$

These truth conditions say that there is no one who knocked without leaving, which is a reasonable approximation in Predicate Logic of the truth conditions of a donkey conditional.

In the absence of donkey anaphora, our lexical entry for *if* increases the potential for spurious ambiguity. For example, we can analyze *someone* in (44) as taking scope either just over *knocked* or just under *if*, with the same truth condition.

(44) If someone knocked, the lobby was open.

This kind of ambiguity is present in all theories that allow wide-scope indefinites to take intermediate scope.

Apart from the lexical entry for *if*, all of the mechanisms for scope and binding were developed entirely independently of any concerns for donkey anaphora!

In fact, even the idea that a lexical item could introduce an outer negation that takes separate scope has been proposed in other empirical domains, though it is controversial (Jacobs 1980, Geurts 1996, de Swart 2000, Penka & Zeijlstra 2005, among others). Most notably, Jacobs (1980) advocates decomposing German *kein* 'no' into negation plus existential quantification in such a way that other quantificational elements can take intermediate scope between the negation and the existential. Geurts (1996) criticizes lexical decomposition as implemented by Jacobs as an unwelcome extension of the expressive power of the formal system. In the current context, however, Geurts' criticism does not apply, since the analysis proposed here for *if* does not rely on any formal resources beyond those already required for basic quantification.

3.1 Multiple indefinites

The prototypical donkey sentence contains more than one indefinite (namely, *a farmer* and *a donkey*). In most treatments of binding, multiple binders are distinguished by means of different subscript indices, e.g., x_i versus x_j. Since our framework is variable-free, we don't use variables, let alone indices!

In our framework, each binder can occupy a different scope-taking level. We don't need to say anything special to achieve this, however, since multiple levels are required independently in order to provide inverse scope, as discussed above in section 2.2. With access to multiple levels, it is easy to handle multiple binders.

Analyzing pronouns as two-level rules is the same thing as claiming that pronouns take scope (see Dowty 2007, who also advocates treating pronouns as scope-takers). Then, roughly speaking, a pronoun chooses its binder by choosing where to take scope. In more practical terms, distinct scope-taking levels correspond to different binders, so layers play the role of indices in some sense: a binder and the pronoun it binds must take effect at the same layer in the compositional tower. One

pleasant consequence of this fact is that (as usual for variable-free treatments) a single lexical entry for each pronoun will do, rather than needing an unbounded number of pronouns distinguished by inaudible indices.

We'll compose the donkey sentence (1a) from its parts, beginning with the antecedent clause. We first apply the function in (27) denoted by the indefinite determiner *a* to each of the two common nouns *farmer* and *donkey*, illustrated below for *farmer*.

(45)
$$\frac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}\Big/\mathrm{N} \quad \mathrm{N} \qquad \frac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}$$
$$\text{a} \quad \text{farmer} \;=\; \text{a farmer}$$
$$\lambda P.\frac{\exists x.\, Px \wedge [\,]}{x} \quad \textbf{farmer} \qquad \frac{\exists x.\, (\textbf{farmer}\,x) \wedge [\,]}{x}$$

We then apply Bind and Lift to each of the two indefinite DPs *a farmer* and *a donkey*, so that they occupy different binding levels.

(46)
$$\overset{\text{Bind}}{\Rightarrow} \begin{array}{c} \mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{DP} \\ \text{a farmer} \\ \exists x.\, (\textbf{farmer}\,x) \wedge ([\,]x) \\ \hline x \end{array} \quad \overset{\text{Lift}}{\Rightarrow} \begin{array}{c} \mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP} \\ \text{a farmer} \\ \exists x.\, (\textbf{farmer}\,x) \wedge ([\,]x) \\ \hline [\,] \\ \hline x \end{array}$$

We build the antecedent clause from three three-level meanings.

(47)
$$\begin{array}{c} \mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP} \\ \textit{a farmer} \\ \exists x.\, (\textbf{farmer}\,x) \wedge ([\,]x) \\ \hline [\,] \\ \hline x \end{array} \quad \begin{array}{c} \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{S}\,|\,\mathrm{S} \\ \hline (\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP} \\ \textit{owns} \\ [\,] \\ \hline [\,] \\ \hline \textbf{owns} \end{array} \quad \begin{array}{c} \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{DP} \\ \textit{a donkey} \\ [\,] \\ \hline \exists y.\, (\textbf{donkey}\,y) \wedge ([\,]y) \\ \hline y \end{array}$$

Next, we build the consequent by Lifting two pronouns from (34), waiting to be bound at two different levels.

(48)
$$\begin{array}{c} \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{DP} \\ \textit{he} \\ \lambda z.\,[\,] \\ \hline [\,] \\ \hline z \end{array} \quad \begin{array}{c} \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline (\mathrm{DP}\backslash\mathrm{S})/\mathrm{DP} \\ \textit{beats} \\ [\,] \\ \hline [\,] \\ \hline \textbf{beats} \end{array} \quad \begin{array}{c} \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP} \\ \textit{it} \\ [\,] \\ \hline \lambda w.\,[\,] \\ \hline w \end{array}$$

We finish the derivation using the lexical entry for *if* given in (41) above (adjusted by an application of Lift).

(49)
$$\begin{array}{c} \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{S}\,|\,\mathrm{S} \\ \hline (\mathrm{S}/\mathrm{S})/\mathrm{S} \\ \textit{if} \\ \neg[\,] \\ \hline [\,] \\ \hline \lambda p\lambda q.\, p \wedge \neg q \end{array} \quad \begin{array}{c} \mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S} \\ \hline \mathrm{S} \\ \textit{a farmer owns a donkey} \\ \exists x.\, (\textbf{farmer}\,x) \wedge ([\,]x) \\ \hline \exists y.\, (\textbf{donkey}\,y) \wedge ([\,]y) \\ \hline \textbf{owns}\,y\,x \end{array} \quad \begin{array}{c} \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{S} \\ \textit{he beats it} \\ \lambda z.\,[\,] \\ \hline \lambda w.\,[\,] \\ \hline \textbf{beats}\,w\,z \end{array}$$

$$= \begin{array}{c} \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{S}\,|\,\mathrm{S} \\ \hline \mathrm{S} \\ \textit{If a farmer owns a donkey he beats it} \\ \neg\exists x.\, (\textbf{farmer}\,x) \wedge ((\lambda z.\,[\,])x) \\ \hline \exists y.\, (\textbf{donkey}\,y) \wedge ((\lambda w.\,[\,])y) \\ \hline (\textbf{owns}\,y\,x) \wedge \neg(\textbf{beats}\,w\,z) \end{array}$$

With two applications of Lower and some routine lambda conversion, we have the following analysis of the standard donkey sentence:

(50) If a farmer owns a donkey, he beats it.
$\neg\exists x.\, (\textbf{farmer}\,x) \wedge \exists y.\, (\textbf{donkey}\,y) \wedge (\textbf{owns}\,y\,x) \wedge \neg(\textbf{beats}\,y\,x)$

These truth conditions require that every farmer beats every donkey that he owns, which is the standard interpretation of the donkey anaphora interpretation of the sentence.

There is much discussion in the literature of so-called 'weak' readings, which require only that each farmer beats at least one of the donkeys that he owns. We follow Barker (1996) and Schein (2003), who argue that weak readings are a pragmatic

domain narrowing effect that do not correspond to a distinct set of truth conditions. Weak versus strong readings for quantificational determiners seem to behave differently (Kanazawa 1994); in section 4, we show how to arrive at either weak or strong readings for each determiner.

3.2 Unwanted uniqueness implications don't arise

The D-type situation-based account first proposed by Heim (1990) (adapting ideas in Berman 1987) and further developed by Elbourne (2005) provides truth conditions for the standard donkey sentence that say, roughly, that every minimal situation of a farmer owning a donkey can be extended to a situation in which the (unique) farmer in that situation beats the (unique) donkey in that situation. These truth conditions are compatible with situations in which a farmer owns more than one donkey, since given a theory of situations like that in Kratzer 1989, we can choose each minimal situation so small that it contains only one farmer and only one donkey (provided we also stipulate suitable persistence behavior, as discussed by Zweig 2006).

However, as pointed out by Kamp (according to the lore in Heim 1990), sometimes even the minimal situation must contain more than one entity that matches the descriptive content of a D-type pronoun.

(51) If a bishop meets a bishop, he blesses him. (= (8))

The problem with (51) is that any situation in which a bishop meets a bishop, no matter how minimal, contains two bishops. If we take the pronoun *he* as a D-type pronoun expressing the content *the bishop* or even *the bishop who meets a bishop*, the uniqueness implication due to the definiteness of the description fails, since there is no unique bishop in any of the relevant minimal situations.

Elbourne (2005: 147) explains how to rescue the situation-based account. The key is to recognize that the compositional semantics treats the two bishops asymmetrically. Under Elbourne's assumptions, quantifier raising creates an LF with the following structure:

(52) [a bishop x [a bishop y [x meets y]$_\gamma$]$_\beta$]$_\alpha$

His truth conditions for interpreting LFs require the existence of a situation γ within which two thin particulars (call them x and y) meet; γ must be contained within a slightly larger situation, β, in which we learn that one of the thin particulars (say, y) happens to be a bishop; and β must be contained within a still larger situation α in which we finally learn that the other thin particular happens to also be a bishop. As Elbourne (2005: 147) puts it, "the inclusion relations among the situations . . . mirror the inclusion relations among the syntactic constituents of the sentence".

We can now construct a semantic property that can tell the bishops apart. Elbourne proposes a property he calls "distinguished", without specifying what that property is, but we can easily choose a suitable property. Let us say that a bishop x is distinguished relative to another bishop y in a given set of situations S just in case there is some $s \in S$ that contains both x and y and the fact that x is a bishop, but not the fact that y is a bishop. Then if S is the set of situations provided by the D-type analysis in the previous paragraph, β will be the member of S that distinguishes one of the bishops.[7]

Elbourne then proposes that "he" and "him" are D-type pronouns both containing the silent NP *bishop*. Pragmatic enrichment then delivers truth conditions equivalent to the definite descriptions *the distinguished bishop* and *the undistinguished bishop*. In summary, Elbourne solves the bishop problem by positing a particular kind of pragmatic enrichment of silent descriptive content.

On the present proposal, of course, bishop sentences are perfectly straightforward and require no special assumptions. The derivation is identical to the one given above for the farmer/donkey sentence (after substituting the appropriate words), giving the following truth conditions (compare with (50) above):

(53) If a bishop meets a bishop, he blesses him.
$\neg\exists x.\,(\mathbf{bishop}\,x) \wedge \exists y.\,(\mathbf{bishop}\,y) \wedge (\mathbf{meets}\,y\,x) \wedge \neg(\mathbf{blesses}\,y\,x)$

Thus bishop sentences pose no special difficulties on our account.

Although donkey pronouns do not entail uniqueness (as we have just seen), Kadmon (1987) and others persistently report that for some speakers donkey pronouns nevertheless contribute a flavor of uniqueness. For instance, many people report that the standard donkey sentences only clearly apply to farmers who own one donkey, and that the status of farmers who own more than one donkey is problematic. Barker (1996) argues that these residual uniqueness effects are due to a presupposition that each farmer either beats all or none of his donkeys. In the absence of a reason to suppose that farmers treat their donkeys uniformly, one way to accommodate the presupposition is to limit attention to farmers that have only one donkey. In contrast, in situations in which the homogeneity presupposition is entailed, all uniqueness implications disappear entirely, as in Heim's famous sage-plant sentence.

(54) Most women who buy a sage plant here buy eight others along with it.

In (54), there is no temptation to claim that the generalization only applies to women who buy a single sage plant.

7 Nick Kroll and Anna Szabolcsi have independently pointed out that something more needs to be said about cases in which the two indefinites accidentally pick out the same individual. For instance, consider *If a scholar cites a scholar, she spells her name carefully* in a situation in which a scholar cites her own work. Perhaps equality among thin particulars, like bishophood, can be omitted in subsituations.

3.3 Extending the account to modal treatments of conditionals

Our analysis approximates the semantics of the conditional using quantification only over individuals. But conditionals display bewildering modal behavior that is not captured in our fragment. It is far from trivial to combine a dynamic account of modality and of binding in a single system (though see Brasoveanu 2007 for a recent example), and we will not work out the details of such an endeavor here. Nevertheless, we give enough details to make it plausible that our overall strategy is compatible with a more nuanced account of conditionals.

One popular strategy for handling conditionals due to Kratzer (e.g., 1991) is to view the *if* clause as a restriction on the accessibility relation of some modal operator. The modal operator in question can be supplied by an adverbial element in the main clause (e.g., *might* in *If a farmer owns a donkey, he might beat it*). If no modal operator is overt, a silent one is assumed to be present. For instance, Heim (1982) suggests that the prototypical donkey sentence (1a) means roughly *If a farmer owns a donkey, he **usually** beats it*.

Once we have a modal operator, the truth condition of the conditional depends on a modal base f and an ordering source g, both supplied by pragmatic context. In the usual treatment, f and g are functions mapping each world to a set of propositions, and propositions, in turn, are modeled as sets of worlds.

We can then use the following semantic value for *if*.

(55)
$$\frac{\lambda w \lambda w'.\, \neg[\,]}{\lambda p \lambda q.\, (w' \in \max(g(w))(\cap(f(w)+p))) \wedge (w' \notin q)}$$

This value maps each evaluation world w onto a set of possible worlds which is constructed according to the following recipe: take the set of propositions provided by $f(w)$. Add the antecedent (p) to that set.[8] Treating propositions as sets of worlds, intersect them. This gives the set of accessible worlds in which the antecedent is true. Next, eliminate all of these worlds that are less than ideal with respect to the partial order induced by the ordering source g applied to w. Now remove all the worlds in which the consequent is true. Finally (the outermost negation, above the line), return the complement of this set.

This complement set will provide the input to the semantics of the governing modal. For instance, we might have either of the following modals:

(56)
$$\frac{\dfrac{S \mid S}{(DP\backslash S)/(DP\backslash S)}}{\lambda f.\, f} \quad \begin{matrix} might \\ \lambda w.\, \exists w'.\, ([\,]\, w\, w') \end{matrix} \qquad\qquad \frac{\dfrac{S \mid S}{(DP\backslash S)/(DP\backslash S)}}{\lambda f.\, f} \quad \begin{matrix} must \\ \lambda w.\, \forall w'.\, ([\,]\, w\, w') \end{matrix}$$

Notice that *might* and *must* differ only in their modal force: *might* existentially quantifies over worlds, and *must* universally quantifies over worlds.

Then we would have the following analysis for *If a donkey eats, it might sleep*:

(57)
$$\frac{\lambda w.\, \exists w'.\, ([\,]\, w\, w')}{\dfrac{\lambda w.\, \lambda w'.\, \neg\exists d.\, (\mathbf{donkey}\, d) \wedge [\,]}{(w' \in \max(g(w))(\cap(f(w)+(\mathbf{eats}\, d)))) \wedge (w' \notin \mathbf{sleep}\, d)}}$$

After lowering (twice) and lambda-conversion, we have

(58)
$$\lambda w.\, \exists w'.\, \neg\exists d.\, (\mathbf{donkey}\, d) \wedge (w' \in \max(g(w))(\cap(f(w)+(\mathbf{eats}\, d)))) \wedge (w' \notin \mathbf{sleep}\, d)$$

If f is an epistemic modal base, and g is a stereotypical ordering source (in which worlds are more ideal if things happen normally), then the prediction is that the sentence will be true in a world w just in case there is some world w' consistent with what we know, and there is no way to pick a donkey d such that

- d eats in w',
- w' is a maximally normal world (at least compared to other worlds compatible with what we know in which a donkey eats), and
- d fails to sleep in w'.

We're ignoring the possibility that there are no donkeys, or that supposing that some donkey eats is incompatible with our knowledge.

In order for this scheme to work, expressions with category S will have to denote sets of worlds, but nothing we have said prevents this. We would also have to adjust the syntax of *if* and the modals to require that modals must take scope over an *if* clause. Other refinements would be necessary, but we will not pursue any here.

What is important for present purposes is that we have arrived at an analysis which exhibits donkey anaphora: the interaction of the modal, the *if*, the indefinite, and the pronoun is such that the donkey that is doing the eating must be the same donkey that is doing the sleeping. In sum, we are not aware of any reason why the approach suggested here would be incompatible with a more refined analysis of conditionals.

8 For instance, for counterfactuals, the + operation would have to start with the antecedent and add as many propositions in $f(w)$ as possible without creating inconsistency; for anankastic conditionals such as *If you want to get to Harlem, you should take the A train*, the + operation will have to be complex in a different way (von Fintel & Iatridou 2005).

3.4 Why does *every* disrupt donkey anaphora?

If a universal occurs in the antecedent, donkey anaphora is no longer possible:

(59) If everyone owns a donkey, it brays.

More precisely, there is no interpretation on which the indefinite takes narrow scope with respect to the universal and still binds the pronoun.

As discussed below in section 7, previous dynamic accounts such as Dynamic Predicate Logic and Dynamic Montague Grammar define *everyone* in terms of static negation, which is stipulated to be dynamically closed, i.e., to block anaphora between an indefinite taking scope inside the negation and a pronoun outside the scope of negation.

Our account needs no such stipulation: the binding relationships follow immediately from getting the scope of the quantifiers right. Recall from (2) that unlike indefinites, the scope of *every* is generally limited to its minimal clause. In this case, the minimal clause is the antecedent. That means that the only possible analysis of the antecedent in (59) must close off the scope of *every* by applying Lower before the antecedent combines with *if*:

(60)
$$\begin{array}{c}\dfrac{S\,|\,S}{S}\\ \textit{Everyone owns a donkey}\\ \dfrac{\forall x.\,\exists y.\,(\mathbf{donkey}\,y)\wedge[\,]}{\mathbf{owns}\,y\,x}\end{array} \quad \overset{\text{Lower}}{\Rightarrow} \quad \begin{array}{c}S\\ \textit{Everyone owns a donkey}\\ \forall x.\,\exists y.\,(\mathbf{donkey}\,y)\wedge(\mathbf{owns}\,y\,x)\end{array}$$

Because eliminating the quantificational level at which *everyone* takes scope also eliminates any other quantifier on the same level and lower levels, whenever the indefinite takes narrow scope with respect to the universal, the scope of the indefinite must also be limited to the antecedent clause.

There is no obvious semantic reason why universals can't take wide scope beyond their minimal clause, so their scope limitations are presumably purely syntactic. Like most leading accounts of donkey anaphora (including Elbourne 2005), we provide no formal mechanism here that bounds the scope-taking of universals.

4 Donkey anaphora from relative clauses

We turn now from donkey anaphora in conditionals to the other classic case of donkey anaphora, which involves indefinites embedded within a relative clause.

(61) Every farmer who owns a donkey beats it. (= (1b))

To provide a lexical entry for quantifiers such as *every*, we proceed by analogy with conditionals. Just as we distinguish taking scope under *if* from taking scope in its antecedent or consequent, we distinguish taking scope under a quantifier from taking scope in its restrictor or nuclear scope.

(62) Every passenger who had no money received a warning note.

(63) Every passenger with no money received a warning note.

(64) An official from every country attended the ceremony.

For example, on the most natural interpretation of (62) and (63), the quantifier *no* takes scope in the restrictor of *every*, and the quantifier *a* takes scope in the nuclear scope of *every*. In the QR approach developed in Heim & Kratzer 1998: 221, the scope of *no* in (63) motivates a proposal (following May) that prepositional phrases have covert clause-like structure involving a semantically vacuous PRO. As long as quantifiers are allowed to take scope at any constituent of type t, (63) will be assigned the same meaning as (62). In (64), the quantifier *every* takes wide scope over *an*.

In contrast to these three examples, we will analyze *a* in (61) as taking scope under *every* yet over its restrictor and nuclear scope. We will also treat linear scope in (65) as an instance of spurious ambiguity in the absence of donkey anaphora, because the same truth condition obtains whether *a* takes scope just in the restrictor of *every*, as in (62) and (63), or also over the nuclear scope of *every*, as in (61).

(65) Every farmer who owns a donkey left.

We refer the reader to other papers (Barker & Shan 2006; Shan & Barker 2006) for detailed analyses of sentences such as (62)–(65) without donkey anaphora. Those analyses are compatible with the machinery for scope and binding in this paper, because they essentially extend it with a natural treatment of relative clauses and prepositional phrases. The end result is that the nominal *farmer who owns a donkey* has two meanings, one where *a* takes scope inside the relative clause and one where it takes wider scope. Both meanings are shown in (66).

(66)
$$\begin{array}{c}N\\ \text{farmer who owns a donkey}\\ \lambda z.\,(\mathbf{farmer}\,z)\wedge\exists y.\,(\mathbf{donkey}\,y)\wedge(\mathbf{owns}\,y\,z)\end{array} \qquad \begin{array}{c}\dfrac{S\,|\,S}{N}\\ \text{farmer who owns a donkey}\\ \dfrac{\exists y.\,(\mathbf{donkey}\,y)\wedge[\,]}{\lambda z.\,(\mathbf{farmer}\,z)\wedge(\mathbf{owns}\,y\,z)}\end{array}$$

By applying the Bind type-shifter to *a donkey*, we can derive another meaning in

which the donkey y is ready to bind a later pronoun.

(67)
$$\dfrac{\mathrm{S} \mid \mathrm{DP} \rhd \mathrm{S}}{\mathrm{N}} \quad \text{farmer who owns a donkey} \quad \dfrac{\exists y.\,(\mathbf{donkey}\,y) \wedge ([\,]y)}{\lambda z.\,(\mathbf{farmer}\,z) \wedge (\mathbf{owns}\,y\,z)}$$

Below we present lexical entries for quantifiers like *every* that take scope over this last meaning and yield the right truth conditions compositionally.

The usual notion of a quantificational determiner is that it takes a nominal and returns a generalized quantifier, so it has the category $\frac{\mathrm{S} \mid \mathrm{S}}{\mathrm{DP}} / \mathrm{N}$. In the present framework, we provide an extra layer to the translation of *every*, as used in the following derivation of (61).

(68)
$$\left(\dfrac{\mathrm{S} \mid \mathrm{S}}{\frac{\mathrm{S} \mid \mathrm{S}}{\mathrm{DP}} / \mathrm{N}} \;\; \dfrac{\mathrm{S} \mid \mathrm{DP} \rhd \mathrm{S}}{\mathrm{N}} \right) \dfrac{\mathrm{DP} \rhd \mathrm{S} \mid \mathrm{S}}{\dfrac{\mathrm{S} \mid \mathrm{S}}{\mathrm{DP} \backslash \mathrm{S}}}$$
$$\left(\textit{every} \;\; \textit{farmer who owns a donkey} \right) \textit{beats it}$$
$$\left(\dfrac{\neg\exists x.\,[\,]}{\lambda P.\,\dfrac{Px \wedge \neg[\,]}{x}} \;\; \dfrac{\exists y.\,(\mathbf{donkey}\,y) \wedge ([\,]\,y)}{\lambda z.\,(\mathbf{farmer}\,z) \wedge (\mathbf{owns}\,y\,z)} \right) \dfrac{\lambda w.\,[\,]}{\dfrac{[\,]}{\mathbf{beats}\,w}}$$

This derivation gives the truth condition

(69) $\neg\exists x \exists y.\,\mathbf{donkey}\,y \wedge ((\mathbf{farmer}\,x \wedge \mathbf{owns}\,y\,x) \wedge \neg(\mathbf{beats}\,y\,x))$,

as desired. The indefinite makes its usual contribution to the compositional truth conditions: on the one hand, it restricts the generalization to farmers who own a donkey; on the other hand, it binds the pronoun in the verb phrase.

This analysis extends to proportional quantificational determiners. For instance, we can assign *most* the denotation

(70)
$$\dfrac{\textsc{most}(\lambda x \lambda p.\,[\,])}{\lambda P.\,\dfrac{Px \wedge (p \vee [\,])}{x}},$$

where MOST is defined by

(71) $\textsc{most}(F) = \big(|\{x : Fxt\}| < 2 \times |\{x : Fxf\}|\big)$,

writing t and f for the two truth values. In the standard treatment, *most* takes as arguments a pair of sets; here, it takes instead a set of pairs such that $\langle x,t \rangle$ is in the set iff x satisfies the restriction, and $\langle x,f \rangle$ is in the set iff x satisfies both the restriction and the nuclear scope (cf. Pietroski's 2006 analysis of quantifiers in terms of Frege-pairs). The net result is that at least half of the entities that satisfy the restriction must also satisfy the nuclear scope. Clearly, we can accommodate any conservative two-place quantificational determiner by replacing MOST with a different function.

The lexical entry for *most* as given delivers weak truth conditions (each farmer in the witness set need only beat one of his donkeys). We can provide strong truth conditions by replacing $p \vee [\,]$ with $p \vee \neg[\,]$ in (70), and $<$ with $>$ in (71).

We note in passing a difference in how indefinites interact with conditionals versus with quantificational determiners like *every*. As (15a) above and (72) below show, an indefinite in the consequent of *if* takes scope immediately under *if* as easily as an indefinite in the antecedent does.

(72) A donkey runs for shelter if it is raining.

As we discuss further in section 7.3, our formal system already generates the attested reading of (72) in which every donkey runs for shelter if it is raining. In contrast, an indefinite in the nuclear scope of *every* cannot take scope immediately under *every*.

(73) a. Every academic witnessed a raucous debate.
b. $*\neg\exists x \exists y.\,\mathbf{academic}\,x \wedge \mathbf{raucous\text{-}debate}\,y \wedge \mathbf{witness}\,y\,x$

(74) ??Every academic enjoys a raucous debate.

It is impossible for the truth condition of (73) to be that every academic witnessed every raucous debate, and it is dubious whether (74) can mean that every academic enjoys every raucous debate. It appears, then, that *a raucous debate* can take either narrow scope in the nuclear scope of *every*, or wide scope beyond *every*, but not in between. Such unavailability of intermediate scope has been observed elsewhere (Fodor & Sag 1982; Kratzer 1998; Lin 2004), but is not accounted for by our formal system.

5 Coordination and donkey anaphora

As noted by Stone (1992), disjunction constitutes a challenge for at least some dynamic theories of anaphora, including DPL.

(75) If a farmer owns a donkey or a goat, he beats it.

The problem for DPL is that the indefinites *a donkey* and *a goat* introduce two distinct discourse referents, neither of which is a suitable antecedent for *it*. This

challenge is hardly insurmountable, but does seem to require assumptions that go beyond the mechanisms provided by the basic theory of DPL.

The D-type analysis has no trouble with disjunction: every situation that verifies the antecedent will either have a donkey in it or a goat, so there will always be a suitable object for the pronoun *it* to describe. The descriptive content of the pronoun will be something neutral between the two descriptions, something like *the animal*, or perhaps *the donkey or goat*.[9] Elbourne (2005: 19) concludes that disjunction provides an argument in favor of the D-type approach over DPL.

Partee & Rooth (1983) suggest allowing phrases like *John or Bill* to introduce new variables. The treatment of generalized disjunction proposed in Barker 2002 achieves the desired result without any stipulation.

(76)
$$\left(A\Big\backslash\frac{\mathrm{S}\,|\,\mathrm{S}}{A}\right)\Big/A \quad \textit{or} \quad \lambda R\lambda L.\,\frac{(\lambda\kappa.(\kappa L)\vee(\kappa R))(\lambda x.[\,])}{x}$$

This lexical entry is polymorphic: A can be any category.[10]

In particular, we can choose $A = \mathrm{DP}$:

(77)
$$\begin{pmatrix}\mathrm{DP}\\ \textit{John}\end{pmatrix}\begin{pmatrix}\left(\mathrm{DP}\Big\backslash\dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}\right)\Big/\mathrm{DP}\\ \textit{or}\end{pmatrix}\begin{pmatrix}\mathrm{DP}\\ \textit{Bill}\end{pmatrix}\begin{pmatrix}\dfrac{\mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}\backslash\mathrm{S}}\\ \textit{called his mother}\end{pmatrix}$$
$$\overset{\text{Bind}}{\Rightarrow}\begin{pmatrix}\dfrac{\mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S}}{\mathrm{DP}}\\ \textit{John or Bill}\end{pmatrix}\begin{pmatrix}\dfrac{\mathrm{DP}\rhd\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}\backslash\mathrm{S}}\\ \textit{called his mother}\end{pmatrix}$$

This derivation gives the truth condition

(78)
$$(\mathbf{called}\,(\mathbf{mother}\,\mathbf{j})\,\mathbf{j})\vee(\mathbf{called}\,(\mathbf{mother}\,\mathbf{b})\,\mathbf{b}).$$

In other words, the disjunction of *John* and *Bill* is perfectly able to serve as the antecedent of a singular pronoun without any stipulation.

9 Leu (2005) argues that in the general case, the content of the D-type pronoun will have to be essentially contentless, roughly *the entity*.

10 There is no restriction to "conjoinable" types, since that is implicit in the claim that disjunction takes scope over an S, i.e., eventually yielding a result of type t. The semantic value is expressed slightly awkwardly as a redex in order to avoid multiple holes [ ]. This awkwardness is purely expository, however; it is perfectly straightforward to compute the denotations of derivations involving *or* using the linear notation from Shan & Barker 2006, in which the value of *or* is $\lambda R\lambda L\lambda\kappa.\,L\kappa\vee R\kappa$.

A similar derivation goes through for donkey anaphora, i.e., when the disjunction is in the antecedent and the pronoun is in the consequent: *If John or Bill left, he called*. In order to bind the pronoun, the disjunction must scope over the consequent, in which case the truth conditions come out equivalent to

(79)
$$((\mathbf{left}\,\mathbf{j})\to(\mathbf{called}\,\mathbf{j}))\wedge((\mathbf{left}\,\mathbf{b})\to(\mathbf{called}\,\mathbf{b})).$$

Crucially, this truth condition requires whoever left to call.

If the disjuncts are indefinites rather than names, we can choose $A = \dfrac{\mathrm{S}\,|\,\mathrm{S}}{\mathrm{DP}}$ (the category of a generalized quantifier) to get appropriate truth conditions for sentences such as *If a farmer owns a donkey or a goat, he beats it*.

Stone and Elbourne claim that similar anaphora can occur with disjoined sentences (as opposed to disjoined DPs):

(80) If Mary hasn't seen John lately, or Ann misses Bill, she calls him.

On the D-type analysis, all that is required is that each minimal situation involved in the antecedent of (80) contains a man. On our system, we simply choose $A = \dfrac{\mathrm{S}\,|\,\mathrm{DP}\rhd\mathrm{S}}{\mathrm{S}}$, which allows each disjoined clause to provide its own binder independently of the other disjunct.

Thus, although disjunction poses difficulties for some specific dynamic theories of meaning, it by no means poses difficulties for all dynamic theories. In particular, it seems to work out fine in the analysis proposed here.

It is a notable virtue of the D-type analysis that it correctly fails to predict donkey anaphora for conjoined DPs.

(81) If a bishop and a bishop meet, he blesses him.

In contrast with the standard bishop sentence above in (53) in which *meet* is used transitively, it is difficult to interpret the pronouns in (81) as taking the indefinites as antecedents. On the D-type account, this contrast is supposed to be because the two indefinites are treated symmetrically in the intransitive case, at least as far as the construction of the situations characterized by the antecedent is concerned. Hence the covert definite descriptions contributed by the pronouns cannot be enriched to distinguish the two bishops, and the example is rendered infelicitous by the failed uniqueness presupposition of the definite descriptions.

In applying our account to (81), there are two points to be made. The first is that the way described above in which disjunction automatically gives rise in effect to a joint discourse referent does not apply to this kind of conjunction. As noted by Elbourne, the *meet* in (81) requires a plural entity as its argument, so this is

sum-forming conjunction, not generalized conjunction. In any case, the conjunction must mark the conjoined DP as syntactically plural, in which case it cannot serve as the antecedent to a singular pronoun.[11] So the conjunction as a whole cannot provide an antecedent for either pronoun.

The second point is to admit that our account predicts that the conjuncts can serve as separate antecedents. But in the general case, this is necessary:

(82) If a woman and a man meet, she asks him for his number.

The D-type account can also explain the availability of anaphora in this case, since the content of each conjunct provides plenty of leverage for the pronouns to distinguish between the participants in the antecedent situation on the basis of their semantic properties.

However, it is possible to find cases in which the conjuncts are semantically distinct, yet the overall situation is symmetric enough that donkey anaphora becomes infelicitous:

(83) a. #If John and Bill meet, he falls asleep.
b. #If a butcher and a baker meet, he pays him.
c. #If a man walking a dog and a woman walking a dog meet, it barks at it.

The D-type and continuations accounts both predict that these sentences have various readings involving donkey anaphora, yet anaphoric interpretations are difficult in a way that we feel is just like the difficulty of (81). Independently of any grammatical theory of anaphora, we need an account of anaphora resolution. Anaphora resolution is a notoriously difficult area of research (see, e.g., Kehler 2002), in part because so many factors seem to influence the process. The difficulty with the sentences in (83) is that they provide no traction for deciding which of the possible anaphoric relations is the intended one. Since even the D-type account needs some principle of this sort to explain the infelicity of the sentences in (83), we suggest that this is all that is necessary to explain any difficulty associated with (81): the various donkey anaphoric analyses are perfectly grammatical, but pragmatically unresolvable.

6 Donkey weak crossover

In our system, binders must be evaluated before the expressions they bind. In the examples considered in this paper, evaluation order corresponds to linear word order. The prediction is that the same mechanism that rules out weak crossover also rules out examples in which a donkey pronoun precedes its antecedent.

(84) (Weak crossover)
a. Everyone$_i$'s mother loves his$_i$ father.
b. *His$_i$ father loves everyone$_i$'s mother.

(85) (Donkey weak crossover) (= (12))
a. If a farmer owns a donkey, he beats it.
b.*?If he owns it, a farmer beats a donkey.

Because our system embodies a default of processing from left to right, we correctly rule in the (a) examples, and correctly rule out the binding in the (b) examples.

Crucially, the same sensitivity to evaluation order obtains when the order of the antecedent and the consequent are reversed.

(86) (Donkey weak crossover) (= (15))
a. A farmer beats a donkey if he owns it.
b. *He beats it if a farmer owns a donkey.

We assume that the order of the clauses does not affect the semantics of the conditional. Then *if* does not differ between (85) and (86), except in one minor syntactic detail, the direction of a slash: $\dfrac{\mathrm{S} \mid \mathrm{S}}{(\mathrm{S}/\mathrm{S})/\mathrm{S}}$ for (85), $\dfrac{\mathrm{S} \mid \mathrm{S}}{(\mathrm{S}\backslash\mathrm{S})/\mathrm{S}}$ for (86).[12]

Most importantly for our purposes here, there is no donkey anaphora derivation of (86b). We take this as strong evidence that donkey anaphora, like all quantificational anaphora, is sensitive to order, favoring a dynamic account. We also take (86) to show that donkey anaphora is independent of the semantics of the conditional, contrary to the predictions of situation-based accounts such as Elbourne 2005.

(87) *His$_i$ father loves most women who have a son$_i$. (= (11b))

Similarly, we correctly predict that an indefinite in a relative clause, such as *a son* in (87), can only bind a donkey pronoun that is evaluated after it.

7 Comparisons with other dynamic accounts

So far we have concentrated on comparisons with Elbourne's D-type analysis as a well-known modern theory of donkey anaphora with broad empirical coverage. We are now ready to compare our system with other explicitly dynamic accounts. We first explain how our system treats binding and scope uniformly by comparing it to Dynamic Montague Grammar. We then discuss the compositional flexibility that results from this uniformity and that extends our empirical coverage.

11 Kanazawa (2001) argues compellingly against Neale's (1990) suggestion that donkey pronouns are semantically number-neutral.

12 There are processing differences between the two clause orders, however. For instance, in order for the outer negation of the conditional to take inverse scope over the existentials in (86a), an inner application of Lift is necessary that was not necessary for the derivation given in section 3.1 for (85a).

7.1 Dynamic Montague Grammar

The dynamic approach to donkey anaphora is deep and rich, including Kamp 1981, Heim 1982, Groenendijk & Stokhof 1989, 1991, and many more. We concentrate here on Groenendijk & Stokhof's (1989) Dynamic Montague Grammar (DMG), since it is a paradigm example of a dynamic and compositional treatment that has some striking similarities to our approach, yet significant technical, empirical, and philosophical differences, as it turns out.

DMG introduces a type-shifter '↑' to turn a static clause meaning q into its dynamic counterpart $\uparrow q$. In this definition, the down operator $^{\vee}$ from intensional logic deals in assignments ('states') rather than worlds.

(88) $$\uparrow q = \lambda p.\, q \wedge {}^{\vee}p$$

The connective ';' conjoins two dynamic sentence meanings.

(89) $$\phi ; \psi = \lambda p.\, \phi({}^{\wedge}\psi(p))$$

For example, *John walks and John talks* translates as

(90) $$(\uparrow \mathbf{walk}(\mathbf{j})) ; (\uparrow \mathbf{talk}(\mathbf{j})) = \lambda p.\, \mathbf{walk}(\mathbf{j}) \wedge \mathbf{talk}(\mathbf{j}) \wedge {}^{\vee}p.$$

An additional type-shifter '↓' extracts a static truth condition from a dynamic sentence meaning.

(91) $$\downarrow \phi = \phi({}^{\wedge}\mathbf{true})$$

For example, applying ↓ to (90) yields the truth condition

(92) $$\mathbf{walk}(\mathbf{j}) \wedge \mathbf{talk}(\mathbf{j}) \wedge {}^{\vee\wedge}\mathbf{true} = \mathbf{walk}(\mathbf{j}) \wedge \mathbf{talk}(\mathbf{j}).$$

These elements of DMG manage the proposition p in (88) and (89) much as the elements of our system manage continuations: roughly, DMG's '↑' corresponds to (a special case of) our LIFT; DMG's ';' corresponds to (a special case of) our combination schema (21) and (likewise) stipulates left-to-right evaluation; and DMG's '↓' corresponds to our LOWER. Indeed, when Groenendijk & Stokhof (1989) write that "we can look upon the propositions which form the extension of a sentence as something giving the truth conditional contents of its possible continuations", they use the word 'continuation' in a sense very similar to the way we use it.

Is DMG a continuation semantics, then? For binding, DMG only lifts clause meanings: from $\mathbf{walks}(\mathbf{j})$ to $\uparrow \mathbf{walks}(\mathbf{j})$. Analogously, for quantification, Montague's PTQ only lifts noun-phrase meanings: from the individual $\mathbf{j}$ to the continuation consumer $\lambda\kappa.\, \kappa(\mathbf{j})$ (Barker 2002). In contrast to DMG and PTQ, our grammar allows lifting any constituent, not just sentences or noun phrases. This uniformity cashes out our claim that the mechanism by which quantifiers find their scopes is one and the same as the mechanism by which pronouns find their antecedents.

7.2 Binding requires scope

Whereas the point of our system is to bring scope and binding together, the point of DMG is to pull scope and binding apart, as is necessitated by the assumption (which we reject) that the indefinites in (1) cannot take scope over their donkey pronouns. DMG thus sports *two* binding strategies. On the one hand, it provides a stock of variables that are evaluated with respect to an assignment function in the usual way. On the other, it provides a separate system in which the role of variables is played by discourse markers, which are evaluated with respect to a state. The translation of *a* mediates between the states and the continuations, so that A_i *man walks* reduces to

(93) $$\lambda p.\, \exists x.\, \mathbf{man}(x) \wedge \mathbf{walk}(x) \wedge \{x/d_i\}^{\vee}p,$$

which is equivalent in DMG to

(94) $$\mathscr{E}d_i\big((\uparrow \mathbf{man}(d_i)) ; (\uparrow \mathbf{walk}(d_i))\big)$$

in terms of the dynamic version $\mathscr{E}$ of the existential quantifier. The static existential quantifier $\exists$ in (93) binds the variable x in the normal way, which supplies an object to the state switcher $\{x/d_i\}$, so that the discourse marker d_i evaluates to that object wherever it occurs (free) in the continuation p. Thus dynamic binding arises from the interaction between a special case of continuations and a special kind of variables.

A famous fact about DMG is that

(95) $$(\mathscr{E}d\phi) ; \psi = \mathscr{E}d(\phi ; \psi).$$

This equation says that, even if a (dynamic) existential quantifier takes scope only over the first element ϕ in a sequence, it can nevertheless bind discourse markers in subsequent elements ψ. Crucially, this equation finds no counterpart in our system. Our only binding operators are $\exists$, $\forall$, and λ, all of which require scope. Thus, in our system, whenever a quantifier binds a pronoun, it takes scope over the pronoun.

But pulling scope and binding apart complicates DMG. DMG provides two kinds of negation, both operating on dynamic clause meanings, that differ in whether an indefinite that takes scope within the negation can nevertheless bind a discourse marker outside the negation: the static negation $\sim_s\phi$ of a dynamic clause meaning ϕ is the closure $\uparrow\downarrow\sim_d\phi$ of the dynamic negation $\sim_d\phi$ of ϕ. In contrast, it is inconceivable on our account for a quantifier that takes scope inside some negation to bind a pronoun outside that negation, because any pronoun outside that negation must also be outside the scope of that quantifier, and a quantifier can only bind a pronoun within its scope. It is for the same reason in section 3.4 that *every* blocks donkey anaphora.

Because DMG allows an existential operator to bind pronouns outside its scope, it must take special steps to ensure that other operators, including negation, block that ability. Our account refrains from the allowing and obviates the blocking.

7.3 Other accounts based on continuations

As we have just seen, ours is not the only account of donkey anaphora that can be viewed as depending to some extent on continuations. The account of de Groote (2006) reformulates DMG in an elegant continuation-based system. He takes the meaning of a clause to be a relation between its *left context*, which contains individuals introduced so far, and its *right context*, a continuation. A clause that introduces a discourse referent does so by adding an individual to the left context and passing the result to the right context. Similarly, Shan (2001) uses *monads*, closely related to continuations (Wadler 1994), to implement a variable-free dynamic semantics and to predict donkey weak crossover.[13]

We are naturally sympathetic to continuation-based theories of donkey anaphora. However, like DMG, neither de Groote's theory nor Shan's lets indefinites, other quantifiers, and pronouns interact in a uniform system of scope taking. Since we argue that binding exploits the same scope-taking mechanisms required for quantification, we believe that a complete understanding of quantificational binding must be situated within a uniform theory of scope. Our greater empirical coverage relies on the fact that this uniformity provides two kinds of compositional flexibility: first, to let an indefinite take existential scope outside an enclosing operator; second, to manipulate the dynamic meaning of a clause as a semantic value.

The first flexibility is introduced in section 2.2, where we use multiple layers of continuation lifting in order to generate inverse scope. For example, to generate the inverse-scope reading of *everyone loves someone*, we use two layers of lifting: *someone* takes effect in the outer layer (higher level) whereas *everyone* takes effect in the inner layer (lower level). Precisely the same machinery turns out to support multiple donkey indefinites (section 3.1) with good results on bishop sentences (section 3.2), and to predict donkey weak crossover (section 6). In particular, we exploit multiple layers of lifting to generate the intended reading of (86a). Similarly, our compositional semantics offers three ways to interpret (96):

(96) A farmer sells lemonade if it is sunny.

It is easy for a dynamic semantics to generate the reading in which *a farmer* takes narrowest scope: the farmers take turns selling lemonade on sunny days. Using multiple layers of lifting, the indefinites in (86a) and (96) can take scope outside their enclosing negation, so we generate two other readings for (96) as desired: the farmers all sell lemonade on sunny days; a certain farmer sells lemonade on sunny days.

13 Like Jacobson (1999), Shan (2001) does not deal with binding out of a possessor (see Barker 2005), whereas we do.

The second flexibility is that a lifted meaning in our system is a semantic value that can be fed to and returned from functions just like any other value. For example, the lexical meaning for *a* in (27) is a function that returns a lifted individual when combined with a common-noun property by ordinary function application. The denotations of *every* in (68) and *or* in (76) also involve functions that return lifted meanings, and we can use *or* to coordinate indefinites and binders as usual. Another example of the second flexibility is how the lexical meaning for *he* in (34) uses $\lambda y.\,[\,]$ on the higher level to abstract over a propositional meaning where indefinites and other quantifiers may take scope (as in *he loves someone*). Such abstractions play a crucial role in our modal meaning for *if* ($\lambda w \lambda w'.\,\neg[\,]$ in (55)) and our proportional meaning for *most* ($\lambda x \lambda p.\,[\,]$ in (70)).

8 Conclusion

The picture that we have drawn is very simple: a quantifier can only bind a pronoun that it takes scope over. But since indefinites can take scope over more than their minimal clause, they can bind donkey pronouns. The fact that they fail to c-command those pronouns is irrelevant, because c-command is not a requirement for binding. However, since evaluation order is a requirement for binding, donkey indefinites must be evaluated before the pronouns that they bind, which in the examples discussed here means they must linearly precede those pronouns. If the pronoun comes first, donkey weak crossover results.

We have presented a formal fragment implementing these ideas that involves three type-shifters that apply freely: Lift, Lower, and Bind. Given independently motivated lexical entries for quantifiers and pronouns, along with an innovative lexical entry for *if*, we derive a wide range of predictions about donkey anaphora.

To be sure, we leave many questions unanswered. For instance, we have limited the discussion above to quantificational binding. What about other types of anaphora? Because our Bind type-shifter applies to proper names such as *John* just as it applies to quantifiers such as *everyone*, the sentence *John*$_i$ *loves his*$_i$ *mother* has a derivation on which *John* grammatically binds *his*. Therefore our system has something to say about at least some non-quantificational anaphora. But there are other cases of non-quantificational anaphora that cannot be handled by our system, such as *He has to offer her an apology if John wants Mary to talk to him again* (cf. the ungrammatical (15b) with quantificational DPs instead of proper names). Our working assumption is that quantificational binding may be subject to different grammatical constraints than anaphora in general. In fact, we suspect that one of the factors that may have lead Reinhart (e.g., 1983) and others to impose what we have argued to be an erroneous c-command requirement on quantificational anaphora was a valiant but ultimately futile attempt to unify quantificational anaphora with

anaphora in general.

We have also limited ourselves to intrasentential binding. But like other dynamic treatments (Heim 1982, Groenendijk & Stokhof 1991, etc.), nothing prevents us from defining a convention for dynamically combining sequences of sentences. And in general, it is easy to imagine how our system could allow an expression to give rise to a side effect in one sentence that affects the evaluation of some expression in a different sentence. But just like our predecessors, we have nothing to say about how to understand what is going on when a side effect persists across different types of speech acts, let alone across different speakers (*Did you see a man$_i$ run past? Yes, he$_i$ turned left.*).

At several points we have emphasized that our theory predicts that relative scope and binding potential depends on the order in which expressions are evaluated. One of the virtues of our system is that it makes it natural for a quantifier to be able to take inverse scope (that is, to get evaluated before some quantifier that linearly precedes it, by occupying a higher level in the tower) without necessarily allowing a pronoun to be evaluated before its binder. But there are well-known cases of quantificational binding in which the pronoun does linearly precede its binder:

(97) Which of his$_i$ relatives does every man$_i$ love _ the most?

Perhaps surprisingly, these so-called reconstruction cases fall out without special stipulation given certain independently motivated assumptions about the syntax and semantics of *wh*-expressions. The basic idea is that, as a result of its syntactic displacement, the evaluation of the fronted *wh*-phrase *which of his relatives* is postponed until after the quantifier *every man* has already been evaluated (Shan & Barker 2006: 123, Barker 2008).

It bears mentioning that although we have argued that donkey pronouns are the most ordinary type of pronouns, and therefore that donkey anaphora does not justify resorting to a D-type analysis, there are many other uses of pronouns that appear to argue in favor of an E-type or D-type analysis.

(98) After everyone dropped his paycheck on the floor, everyone picked it up again.

There is a salient reading of (98) on which the pronoun appears to mean something like *his paycheck*, with the virtual *his* bound by *everyone*. As long as we generalize the lexical entry for pronouns as in Shan & Barker 2006 and Barker 2008, our system is perfectly able to bind into a functional pronoun in the style of, e.g., Jacobson 1999. However, it remains as much a mystery for us as for everyone else how that functional pronoun comes to have the content of the **paycheck** function.

None of these open questions diminishes our belief that understanding quantificational binding requires a theory that unifies scope-taking with binding. We have proposed just such a theory, one based on continuations, which allows expressions to dynamically manipulate their semantic context. To what degree this approach will provide useful insights into further mysteries remains to be discovered.

9 Appendix: converting to lambda terms

This appendix illustrates how to convert semantic towers into lambda terms in the linear notation of Shan & Barker 2006. Recall that $\frac{f[\,]}{x}$ is shorthand for $\lambda\kappa.\, f[\kappa(x)]$. Accordingly, our type shifters correspond to lambda terms as follows.

(23) $\mathbf{L} = \text{Lift} = \lambda x.\, \lambda\kappa.\, \kappa x$

(24) $\text{Lower} = \lambda F.\, F(\lambda x.\, x)$

(36) $\mathbf{B} = \text{Bind} = \lambda X.\, \lambda\kappa. X(\lambda x.\, \kappa x x)$

These type shifters are called Up, Down, and Bind in Shan & Barker 2006. Tower combination can also be expressed in linear notation, whether the lifted function (f) finds its lifted argument (x) to the right ($\mathbf{S}_/$) or to the left ($\mathbf{S}_\backslash$):

(21) $\mathbf{S}_/ = \lambda L.\, \lambda R.\, \lambda\kappa.\, L(\lambda f.\, R(\lambda x.\, \kappa(fx)))$

$\mathbf{S}_\backslash = \lambda L.\, \lambda R.\, \lambda\kappa.\, L(\lambda x.\, R(\lambda f.\, \kappa(fx)))$

In Shan & Barker 2006, $\mathbf{S}_/$ is posited as a primitive called Scope, whereas $\mathbf{S}_\backslash$ can be derived as Scope $\circ$ (Scope (Up Lift)), where $\circ$ denotes function composition and Lift is as defined in that paper.

Thus we can compute the semantics of the donkey sentence from (42) as follows.

(42)
$$\begin{aligned}
\mathbf{B}\ \text{someone} &= \lambda\kappa.\ \exists y.\ \kappa y y\\
\mathbf{L}\ \text{knocked} &= \lambda\kappa.\ \kappa(\mathbf{knocked})\\
\mathbf{S}_{\backslash}\ (\mathbf{B}\ \text{someone})\ (\mathbf{L}\ \text{knocked}) &= \lambda\kappa.\ \exists y.\ \kappa(\mathbf{knocked}\,y)y\\
\text{she} &= \lambda\kappa.\ \lambda x.\ \kappa x\\
\mathbf{L}\ \text{left} &= \lambda\kappa.\ \kappa(\mathbf{left})\\
\mathbf{S}_{\backslash}\ \text{she}\ (\mathbf{L}\ \text{left}) &= \lambda\kappa.\ \lambda x.\ \kappa(\mathbf{left}\,x)\\
\text{if} &= \lambda\kappa.\ \neg\kappa(\lambda p\lambda q.\ p\wedge\neg q)\\
\mathbf{S}_{/}\ \text{if} &= \lambda R.\ \lambda\kappa.\ \neg R(\lambda p.\ \kappa(\lambda q.\ p\wedge\neg q))
\end{aligned}$$

$$\mathbf{S}_{/}\ \text{if}\ (\mathbf{S}_{\backslash}\ (\mathbf{B}\ \text{someone})\ (\mathbf{L}\ \text{knocked})) = \lambda\kappa.\ \neg\exists y.\ \kappa(\lambda q.\ (\mathbf{knocked}\,y)\wedge\neg q)y$$

$$\mathbf{S}_{/}\ (\mathbf{S}_{/}\ \text{if}\ (\mathbf{S}_{\backslash}\ (\mathbf{B}\ \text{someone})\ (\mathbf{L}\ \text{knocked})))\ (\mathbf{S}_{\backslash}\ \text{she}\ (\mathbf{L}\ \text{left})) = \lambda\kappa.\ \neg\exists y.\ \kappa((\mathbf{knocked}\,y)\wedge\neg(\mathbf{left}\,y))$$

$$\text{Lower}\ (\mathbf{S}_{/}\ (\mathbf{S}_{/}\ \text{if}\ (\mathbf{S}_{\backslash}\ (\mathbf{B}\ \text{someone})\ (\mathbf{L}\ \text{knocked})))\ (\mathbf{S}_{\backslash}\ \text{she}\ (\mathbf{L}\ \text{left}))) = \neg\exists y.\ (\mathbf{knocked}\,y)\wedge\neg(\mathbf{left}\,y)$$

Checking each step in this derivation is a good way to appreciate the convenience of the tower notation.

References

Barker, Chris. 1996. Presuppositions for proportional quantifiers. *Natural Language Semantics* 4(3): 237–259. doi:10.1007/BF00372821.

Barker, Chris. 2002. Continuations and the nature of quantification. *Natural Language Semantics* 10(3): 211–242. doi:10.1023/A:1022183511876.

Barker, Chris. 2005. Remark on Jacobson 1999: Crossover as a local constraint. *Linguistics and Philosophy* 28(4): 447–472. doi:10.1007/s10988-004-5327-1.

Barker, Chris. 2008. Reconstruction as delayed evaluation. In Erhard Hinrichs & John Nerbonne (eds.) *Theory and Evidence in Semantics*. CSLI Publications.

Barker, Chris & Chung-chieh Shan. 2006. Types as graphs: Continuations in type logical grammar. *Journal of Logic, Language and Information* 15(4): 331–370. doi:10.1007/s10849-006-0541-6.

Berman, Stephen. 1987. Situation-based semantics for adverbs of quantification. In James Blevins & Anne Vainikka (eds.) *Studies in Semantics*, *University of Massachusetts Occasional Papers in Linguistics (UMOP)*, vol. 12, 46–68. GLSA, University of Masschusetts at Amherst.

Brasoveanu, Adrian. 2007. *Structured Nominal and Modal Reference*. Ph.D. thesis, Rutgers University.

Büring, Daniel. 2004. Crossover situations. *Natural Language Semantics* 12(1): 23–62. doi:10.1023/B:NALS.0000011144.81075.a8.

Chierchia, Gennaro. 1995. *Dynamics of Meaning: Anaphora, Presupposition, and the Theory of Grammar*. University of Chicago Press.

Dowty, David. 2007. Compositionality as an empirical problem. In Chris Barker & Pauline Jacobson (eds.) *Direct Compositionality*. Oxford University Press.

Elbourne, Paul. 2005. *Situations and Individuals*. MIT Press.

Evans, Gareth. 1977. Pronouns, quantifiers, and relative clauses (I). *Canadian Journal of Philosophy* 7(3): 467–536. Reprinted in Mark Platts (ed). 1980. *Reference, Truth and Reality: Essays on the Philosophy of Language*. Also reprinted in Gareth Evans. 1985. *Collected Papers*. Clarendon, Oxford. 76–152. Page numbers in the text refer to the 1985 reprinting.

Evans, Gareth. 1980. Pronouns. *Linguistic Inquiry* 11(2): 337–362.

Farkas, Donka. 1981. Quantifier scope and syntactic islands. *Chicago Linguistics Society* 17: 59–66.

von Fintel, Kai & Sabine Iatridou. 2005. What to do if you want to go to Harlem: Anankastic conditionals and related matters. Ms, MIT.

Fodor, Janet Dean & Ivan A. Sag. 1982. Referential and quantificational indefinites. *Linguistics and Philosophy* 5(3): 355–398. doi:10.1007/BF00351459.

Geurts, Bart. 1996. On *No*. *Journal of Semantics* 13(1): 67–86. doi:10.1093/jos/13.1.67.

Groenendijk, Jeroen & Martin Stokhof. 1989. Dynamic Montague grammar. In L. Kalman & L. Polos (eds.) *Proceedings of the Second Symposium on Logic and Language*, 3–48. Eotvos Lorand University Press.

Groenendijk, Jeroen & Martin Stokhof. 1991. Dynamic predicate logic. *Linguistics and Philosophy* 14(1): 39–100. doi:10.1007/BF00628304.

de Groote, Philippe. 2006. Towards a Montogovian account of dynamics. *Proceedings of Semantics and Linguistic Theory* 16.

Heim, Irene. 1982. *The Semantics of Definite and Indefinite Noun Phrases*. Ph.D. thesis, University of Massachusetts at Amherst.

Heim, Irene. 1990. E-type pronouns and donkey anaphora. *Linguistics and Philosophy* 13(2): 137–177. doi:10.1007/BF00630732.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Oxford: Blackwell.

Jacobs, Joachim. 1980. Lexical decomposition in Montague grammar. *Theoretical Linguistics* 7: 121–136.

Jacobson, Pauline. 1999. Towards a variable-free semantics. *Linguistics and Philosophy* 22(2): 117–185. doi:10.1023/A:1005464228727.

Kadmon, Nirit. 1987. *On Unique and Non-Unique Reference and Asymmetric Quantification*. Ph.D. thesis, University of Massachusetts at Amherst.

Kamp, Hans. 1981. A theory of truth and semantic interpretation. In Jeroen Groenendijk, Theo Janssen & Martin Stokhof (eds.) *Formal Methods in the*

Study of Language, 277–322. Mathematical Centre Amsterdam.

Kanazawa, Makoto. 1994. Weak vs. strong readings of donkey sentences and monotonicity inference in a dynamic setting. *Linguistics and Philosophy* 17(2): 109–158. doi:10.1007/BF00984775.

Kanazawa, Makoto. 2001. Singular donkey pronouns are semantically singular. *Linguistics and Philosophy* 24(3): 383–403. doi:10.1023/A:1010766724907.

Kehler, Andrew. 2002. *Coherence, Reference, and the Theory of Grammar*. CSLI Publications.

Kratzer, Angelika. 1989. An investigation of the lumps of thought. *Linguistics and Philosophy* 12(5): 607–653. doi:10.1007/BF00627775.

Kratzer, Angelika. 1991. Modality. In Arnim von Stechow & Dieter Wunderlich (eds.) *Semantics: An International Handbook of Contemporary Research*, 639–650. Berlin: de Gruyter.

Kratzer, Angelika. 1998. Scope or pseudoscope? Are there wide-scope indefinites? In Susan Rothstein (ed.) *Events and Grammar*. Dordrecht: Kluwer.

Leu, Tom. 2005. Donkey pronouns: Void descriptions? *Proceedings of the North East Linguistics Society* 35.

Lin, Jo-wang. 2004. Choice functions and scope of existential polarity wh-phrases in Mandarin Chinese. *Linguistics and Philosophy* 27(4): 451–491. doi:10.1023/B:LING.0000024407.76999.f7.

May, Robert. 1977. *The Grammar of Quantification*. Ph.D. thesis, Massachusetts Institute of Technology.

Neale, Stephen. 1990. *Descriptions*. MIT Press.

Partee, Barbara H. & Mats Rooth. 1983. Generalized conjunction and type ambiguity. In Rainer Bäuerle, Christoph Schwarze & Arnim von Stechow (eds.) *Meaning, Use, and Interpretation of Language*, 361–383. de Gruyter.

Penka, Doris & Hedde Zeijlstra. 2005. Negative indefinites in Dutch and German. URL http://ling.auf.net/lingbuzz/000192. Ms, University of Tübingen.

Pietroski, Paul. 2006. Interpreting concatenation and concatenates. *Philosophical Issues* 16: 221–245. doi:10.1111/j.1533-6077.2006.00111.x.

Plotkin, G. D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1(2): 125–159. doi:10.1016/0304-3975(75)90017-1.

Reinhart, Tanya. 1983. *Anaphora and Semantic Interpretation*. London: Croom Helm.

Reinhart, Tanya. 1997. Quantifier scope: How labor is divided between QR and choice functions. *Linguistics and Philosophy* 20(4): 335–397. doi:10.1023/A:1005349801431.

Schein, Barry. 2003. Adverbial, descriptive reciprocals. *Philosophical Perspectives* 17(1): 333. doi:10.1111/j.1520-8583.2003.00013.x.

Schwarzschild, Roger. 2002. Singleton indefinites. *Journal of Semantics* 19(3): 289–314. doi:10.1093/jos/19.3.289.

Shan, Chung-chieh. 2001. A variable-free dynamic semantics. *Proceedings of the Amsterdam Colloquium* 13: 204–209.

Shan, Chung-chieh. 2005. *Linguistic Side Effects*. Ph.D. thesis, Harvard University.

Shan, Chung-chieh & Chris Barker. 2006. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy* 29(1): 91–134. doi:10.1007/s10988-005-6580-7.

Steedman, Mark. 2000. *The Syntactic Process*. MIT Press.

Stone, Matthew. 1992. *Or* and anaphora. *Proceedings of Semantics and Linguistic Theory* 2: 367–385.

de Swart, Henriëtte. 2000. Scope ambiguities with negative quantifiers. In Klaus von Heusinger & Urs Egli (eds.) *Reference and Anaphoric Relations*, 118–142. Kluwer.

Szabolcsi, Anna. 2007. Scope and binding. To appear in Klaus von Heusinger, Claudia Maidenborn, and Paul Portner (eds). *Handbook of Semantics: An international handbook of natural language meaning*. Mouton de Gruyter, Berlin.

Wadler, Philip. 1994. Monads and composable continuations. *LISP and Symbolic Computation* 7(1): 39–55. doi:10.1007/BF01019944.

Zweig, Eytan. 2006. When the donkey lost its fleas: Persistence, minimal situations, and embedded quantifiers. *Natural Language Semantics* 14(4): 283–296. doi:10.1007/s11050-006-9003-6.

Finally Tagless, Partially Evaluated*

Tagless Staged Interpreters for Simpler Typed Languages

Jacques Carette[1], Oleg Kiselyov[2], and Chung-chieh Shan[3]

[1] McMaster University carette@mcmaster.ca
[2] FNMOC oleg@pobox.com
[3] Rutgers University ccshan@rutgers.edu

Abstract. We have built the first family of tagless interpretations for a higher-order typed object language in a typed metalanguage (Haskell or ML) that require no dependent types, generalized algebraic data types, or postprocessing to eliminate tags. The statically type-preserving interpretations include an evaluator, a compiler (or staged evaluator), a partial evaluator, and call-by-name and call-by-value CPS transformers.

Our main idea is to encode HOAS using cogen functions rather than data constructors. In other words, we represent object terms not in an initial algebra but using the coalgebraic structure of the λ-calculus. Our representation also simulates inductive maps from types to types, which are required for typed partial evaluation and CPS transformations.

Our encoding of an object term abstracts over the various ways to interpret it, yet statically assures that the interpreters never get stuck. To achieve self-interpretation and show Jones-optimality, we relate this exemplar of higher-rank and higher-kind polymorphism to plugging a term into a context of let-polymorphic bindings.

> It should also be possible to define languages with a highly refined syntactic type structure. Ideally, such a treatment should be metacircular, in the sense that the type structure used in the defined language should be adequate for the defining language.
>
> John Reynolds [28]

1 Introduction

A popular way to define and implement a language is to embed it in another [28]. Embedding means to represent terms and values of the *object language* as terms and values in the *metalanguage*. Embedding is especially appropriate for domain-specific object languages because it supports rapid prototyping and integration with the host environment [16]. If the metalanguage supports *staging*, then the embedding can compile object programs to the metalanguage and avoid the overhead of interpreting them on the fly [23]. A staged definitional interpreter is thus a promising way to build a domain-specific language (DSL).

* We thank Martin Sulzmann and Walid Taha for helpful discussions. Sam Staton, Pieter Hofstra, and Bart Jacobs kindly provided some useful references. We thank anonymous reviewers for pointers to related work.

$$\frac{\begin{matrix}[x:t_1]\\ \vdots \\ e:t_2\end{matrix}}{\lambda x.\, e : t_1 \to t_2} \qquad \frac{\begin{matrix}[f:t_1\to t_2]\\ \vdots \\ e:t_1 \to t_2\end{matrix}}{\text{fix}\, f.\, e : t_1 \to t_2} \qquad \frac{e_1 : t_1 \to t_2 \quad e_2 : t_1}{e_1 e_2 : t_2} \qquad \frac{n \text{ is an integer}}{n : \mathbb{Z}}$$

$$\frac{b \text{ is a boolean}}{b : \mathbb{B}} \qquad \frac{e:\mathbb{B} \quad e_1 : t \quad e_2 : t}{\text{if } e \text{ then } e_1 \text{ else } e_2 : t} \qquad \frac{e_1:\mathbb{Z} \quad e_2:\mathbb{Z}}{e_1 + e_2 : \mathbb{Z}} \qquad \frac{e_1:\mathbb{Z} \quad e_2:\mathbb{Z}}{e_1 \times e_2 : \mathbb{Z}} \qquad \frac{e_1:\mathbb{Z} \quad e_2:\mathbb{Z}}{e_1 \le e_2 : \mathbb{B}}$$

Fig. 1. Our typed object language

We focus on embedding a *typed* object language into a *typed* metalanguage. The benefit of types in this setting is to rule out meaningless object terms, thus enabling faster interpretation and assuring that our interpreters do not get stuck. To be concrete, we use the typed object language in Figure 1 throughout this paper. We aim not just for evaluation of object programs but also for compilation, partial evaluation, and other processing.

Pašalić et al. [23] and Xi et al. [37] motivated interpreting a typed object language in a typed metalanguage as an interesting problem. The common solutions to this problem store object terms and values in the metalanguage in a universal type, a generalized algebraic data type (GADT), or a dependent type. In the remainder of this section, we discuss these solutions, identify their drawbacks, then summarize our proposal and contributions. We leave aside the solved problem of writing a parser/type-checker, for embedding object language objects into the metalanguage (whether using dependent types [23] or not [2]), and just enter them by hand.

1.1 The tag problem

It is straightforward to create an algebraic data type, say in OCaml, Fig. 2(a), to represent object terms such as those in Figure 1. For brevity, we elide treating integers, conditionals, and fixpoint in this section. We represent each variable using a unary de Bruijn index.[4] For example, we represent the object term $(\lambda x.\, x)$ true as `let test1 = A (L (V VZ), B true)`.

```
(a) type var = VZ | VS of var
    type exp = V of var | B of bool | L of exp | A of exp * exp
(b) let rec lookup (x::env) = function VZ -> x | VS v -> lookup env v
    let rec eval0 env = function
    | V v       -> lookup env v
    | B b       -> b
    | L e       -> fun x -> eval0 (x::env) e
    | A (e1,e2) -> (eval0 env e1) (eval0 env e2)
(c) type u = UB of bool | UA of (u -> u)
(d) let rec eval env = function
    | V v       -> lookup env v
    | B b       -> UB b
    | L e       -> UA (fun x -> eval (x::env) e)
    | A (e1,e2) -> match eval env e1 with UA f -> f (eval env e2)
```

Fig. 2. OCaml code illustrating the tag problem

[4] We use de Bruijn indices to simplify the comparison with Pašalić et al.'s work [23].

Following [23], we try to implement an interpreter function `eval0`, Fig. 2(b). It takes an object term such as `test1` above and gives us its value. The first argument to `eval0` is the environment, initially empty, which is the list of values bound to free variables in the interpreted code. If our OCaml-like metalanguage were untyped, the code above would be acceptable. The `L e` line exhibits interpretive overhead: `eval0` traverses the function body `e` every time (the result of evaluating) `L e` is applied. Staging can be used to remove this interpretive overhead [23, §1.1–2].

However, the function `eval0` is ill-typed if we use OCaml or some other typed language as the metalanguage. The line `B b` says that `eval0` returns a boolean, whereas the next line `L e` says the result is a function, but all branches of a pattern-match form must yield values of the same type. A related problem is the type of the environment `env`: a regular OCaml list cannot hold both boolean and function values.

The usual solution is to introduce a universal type [23, §1.3] containing both booleans and functions, Fig. 2(c). We can then write a typed interpreter, Fig. 2(d), whose inferred type is `u list -> exp -> u`. Now we can evaluate `eval [] test1` obtaining `UB true`. The unfortunate tag `UB` in the result reflects that `eval` is a partial function. First, the pattern match `with UA f` in the line `A (e1,e2)` is not exhaustive, so `eval` can fail if we apply a boolean, as in the ill-typed term `A (B true, B false)`. Second, the `lookup` function assumes a nonempty environment, so `eval` can fail if we evaluate an open term `A (L (V (VS VZ)), B true)`. After all, the type `exp` represents object terms both well-typed and ill-typed, both open and closed.

If we evaluate only closed terms that have been type-checked, then `eval` would never fail. Alas, this soundness is not obvious to the metalanguage, whose type system we must still appease with the nonexhaustive pattern matching in `lookup` and `eval` and the tags `UB` and `UA` [23, §1.4]. In other words, the algebraic data types above fail to express in the metalanguage that the object program is well-typed. This failure necessitates tagging and nonexhaustive pattern-matching operations that incur a performance penalty in interpretation [23] and impair optimality in partial evaluation [33]. In short, the universal-type solution is unsatisfactory because it does not preserve typing.

It is commonly thought that to interpret a typed object language in a typed metalanguage while preserving types is difficult and requires GADTs or dependent types [33]. In fact, this problem motivated much work on GADTs [24, 37] and on dependent types [11, 23]. Yet other type systems have been proposed to distinguish closed terms like `test1` from open terms [9, 21, 34], so that `lookup` never receives an empty environment. We discuss these proposals further in §5.

1.2 Our final proposal

We represent object programs using ordinary functions rather than data constructors. These functions comprise the entire interpreter, shown below.

```
let varZ env    = fst env          let b (bv:bool) env = bv
let varS vp env = vp (snd env)     let lam e env   = fun x -> e (x,env)
let app e1 e2 env   = (e1 env) (e2 env)
```

We now represent our sample term $(\lambda x.\, x)$ true as `let testf1 = app (lam varZ) (b true)`. This representation is almost the same as in §1.1, only written with lowercase identifiers. To evaluate an object term is to apply its representation to the empty environment, `testf1 ()`, obtaining `true`. The result has no tags: the interpreter patently uses no tags and no pattern matching. The term `b true` evaluates to a boolean and the term `lam varZ` evaluates to a function, both untagged. The `app` function applies `lam varZ` without pattern matching. What is more, evaluating an open term such as `app (lam (varS varZ)) (b true)` gives a type error rather than a run-time error. The type error correctly complains that the initial environment should be a tuple rather than `()`. In other words, the term is open.

In sum, by Church-encoding terms using ordinary functions, we achieve a tagless evaluator for a typed object language in a metalanguage with a simple Hindley-Milner type system. In this *final* rather than *initial* approach, both kinds of run-time errors in §1.1 (applying a nonfunction and evaluating an open term) are reported at compile time. Because the new interpreter uses no universal type or pattern matching, it never results in a run-time error, and is in fact total. Because this safety is obvious not just to us but also to the metalanguage implementation, we avoid the serious performance penalty [23] of error checking. Glück [12] explains deeper technical reasons that inevitably lead to these performance penalties.

Our solution is *not* Church-encoding the universal type. The Church encoding of the type `u` in §1.1 requires two continuations; the function `app` in the interpreter above would have to provide both to the encoding of `e1`. The continuation corresponding to the `UB` case of `u` must either raise an error or loop. For a well-typed object term, that error continuation is never invoked, yet it must be supplied. In contrast, our interpreter has no error continuation at all.

The evaluator above is wired directly into the functions `b`, `lam`, `app`, and so on. We explain how to abstract the interpreter so as to process the *same* term in many other ways: compilation, partial evaluation, CPS conversion, and so forth.

1.3 Contributions

The term "constructor" functions `b`, `lam`, `app`, and so on appear free in the encoding of an object term such as `testf1` above. Defining these functions differently gives rise to different interpreters, that is, different folds on object programs. Given the same term representation but varying the interpreter, we can

- evaluate the term to a value in the metalanguage;
- measure the size or depth of the term;
- compile the term, with staging support such as in MetaOCaml;
- partially evaluate the term, online; and
- transform the term to continuation-passing style (CPS), even call-by-name (CBN) CPS, so as to isolate the evaluation order of the object language from that of the metalanguage.[5]

[5] Due to serious lack of space, we refer the reader to the accompanying code for this.

We have programmed our interpreters in OCaml (and, for staging, MetaOCaml [19]) and standard Haskell. The complete code is available at `http://okmij.org/ftp/packages/tagless-final.tar.gz` to supplement the paper. For simplicity, main examples in the paper will be in MetaOCaml; all examples have also been implemented in Haskell.

We attack the problem of tagless (staged) typed-preserving interpretation exactly as it was posed by Pašalić et al. [23] and Xi et al. [37]. We use their running examples and achieve the result they call desirable. Our contributions are as follows.

1. We build interpreters that evaluate (§2), compile (or evaluate with staging) (§3), and partially evaluate (§4) a typed higher-order object language in a typed metalanguage, in direct and continuation-passing styles.
2. All these interpreters use no type tags, patently never get stuck, and need no advanced type-system features such as GADTs, dependent types, or intentional type analysis.
3. The partial evaluator avoids polymorphic lift and delays binding-time analysis. It bakes a type-to-type map into the interpreter interface to eliminate the need for GADTs and thus remain portable across Haskell 98 and ML.
4. We use the type system of the metalanguage to check statically that an object program is well-typed and closed.
5. We show clean, comparable implementations in MetaOCaml and Haskell.
6. We specify a functor signature that encompasses all our interpreters, from evaluation and compilation (§2) to partial evaluation (§4).
7. We point a clear way to extend the object language with more features such as state.[6]
8. We describe an approach to self-interpretation compatible with the above. Self-interpretation turned out to be harder than expected.[6]

Our code is surprisingly simple and obvious in hindsight, but it has been cited as a difficult problem ([32] notwithstanding) to interpret a typed object language in a typed metalanguage without tagging or type-system extensions. For example, Taha et al. [33] say that “expressing such an interpreter in a statically typed programming language is a rather subtle matter. In fact, it is only recently that some work on programming type-indexed values in ML [38] has given a hint of how such a function can be expressed.” We discuss related work in §5.

To reiterate, we do *not* propose any new language feature or new technique. We use features already present in mainstream functional languages—Hindley-Milner type system with either an inference-preserving module system or constructor classes, as realized in ML and Haskell 98—and techniques which have all appeared in the literature (in particular, [32, 38]), to solve a problem that was stated in the published record as unsolved and likely unsolvable in ML or Haskell 98 without extensions. The simplicity of our solution and its use of only mainstream features make it more practical to build typed, embedded DSLs.

[6] Again, please see our code.

2 The object language and its tagless interpreters

Figure 1 shows our object language, a simply-typed λ-calculus with fixpoint, integers, booleans, and comparison. The language is close to Xi et al.'s [37], without their polymorphic lift but with more constants so as to more conveniently express Fibonacci, factorial, and power. In contrast to §1, we encode binding using higher-order abstract syntax (HOAS) [20, 25] rather than de Bruijn indices. This makes the encoding convenient and ensures that our object programs are closed.

2.1 How to make encoding flexible: abstract the interpreter

We embed our language in (Meta)OCaml and Haskell. In Haskell, the functions that construct object terms are methods in a type class `Symantics` (with a parameter `repr` of kind `* -> *`), Fig. 3(a). The class is so named because its interface gives the syntax of the object language and its instances give the semantics. For example, we encode the term `test1`, or $(\lambda x.\,x)$ true, from §1.1 above as `app (lam (\x -> x)) (bool True)`, whose inferred type is `Symantics repr => repr Bool`. For another example, the classical *power* function is in Fig. 3(b) and the partial application $\lambda x.\,power\ x\ 7$ is in Fig. 3(c). The dummy argument `()` above is to avoid the monomorphism restriction, to keep the type of `testpowfix` and `testpowfix7` polymorphic in `repr`. The methods `add`, `mul`, and `leq` are quite similar, and so are `int` and `bool`. Therefore, we often show only one method of each group and elide the rest. The accompanying code has the complete implementations.

```
(a) class Symantics repr where
      int :: Int  -> repr Int;       bool :: Bool -> repr Bool
      lam :: (repr a -> repr b) -> repr (a -> b)
      app :: repr (a -> b) -> repr a -> repr b
      fix :: (repr a -> repr a) -> repr a

      add :: repr Int -> repr Int -> repr Int
      mul :: repr Int -> repr Int -> repr Int
      leq :: repr Int -> repr Int -> repr Bool
      if_ :: repr Bool -> repr a -> repr a -> repr a
(b) testpowfix () = lam (\x -> fix (\self -> lam (\n ->
                      if_ (leq n (int 0)) (int 1)
                          (mul x (app self (add n (int (-1))))))))
(c) testpowfix7 () = lam (\x -> app (app (testpowfix ()) x) (int 7))
```

Fig. 3. Symantics in Haskell

Comparing `Symantics` with Fig. 1 shows how to represent *every* typed, closed object term in the metalanguage. Moreover, the representation preserves types.

Proposition 1. *If an object term has the object type t, then its representation in the metalanguage has the type* `forall repr. Symantics repr => repr` *t.*

Conversely, the type system of the metalanguage statically checks that the represented object term is well-typed and closed. If we err, say replace `int 7` with `bool True` in `testpowfix7`, Haskell will complain there that the expected type `Int` does not match the inferred `Bool`. Similarly, the object term $\lambda x.\,xx$ and

```
module type Symantics = sig type ('c, 'dv) repr
  val int : int  -> ('c, int) repr
  val bool: bool -> ('c, bool) repr

  val lam : (('c, 'da) repr -> ('c, 'db) repr) -> ('c, 'da -> 'db) repr
  val app : ('c, 'da -> 'db) repr -> ('c, 'da) repr -> ('c, 'db) repr
  val fix : ('x -> 'x) -> (('c, 'da -> 'db) repr as 'x)

  val add : ('c, int) repr -> ('c, int) repr -> ('c, int) repr
  val mul : ('c, int) repr -> ('c, int) repr -> ('c, int) repr
  val leq : ('c, int) repr -> ('c, int) repr -> ('c, bool) repr
  val if_ : ('c, bool) repr
            -> (unit -> 'x) -> (unit -> 'x) -> (('c, 'da) repr as 'x)
end
module EX(S: Symantics) = struct open S
  let test1 () = app (lam (fun x -> x)) (bool true)
  let testpowfix () =
       lam (fun x -> fix (fun self -> lam (fun n ->
        if_ (leq n (int 0)) (fun () -> int 1)
            (fun () -> mul x (app self (add n (int (-1))))))))
  let testpowfix7 = lam (fun x -> app (app (testpowfix ()) x) (int 7))
end
```

Fig. 4. A simple (Meta)OCaml embedding of our object language, and examples

its encoding `lam (\x -> app x x)` both fail occurs-checks in type checking. Haskell's type checker also flags syntactically invalid object terms, such as if we forget `app` somewhere above.

To embed the same object language in (Meta)OCaml, we replace the type class `Symantics` and its instances by a module signature `Symantics` and its implementations. Figure 4 shows a simple signature that suffices until §4. The two differences are: the additional type parameter `'c`, an *environment classifier* [34] required by MetaOCaml for code generation in §3; and the η-expanded type for `fix` and thunk types in `if_` since OCaml is a call-by-value language.

The functor `EX` in Fig. 4 encodes our running examples `test1` and the *power* function (`testpowfix`). The dummy argument to `test1` and `testpowfix` is an artifact of MetaOCaml, related to monomorphism: in order for us to run a piece of generated code, it must be polymorphic in its environment classifier (the type variable `'c` in Figure 4). The value restriction dictates that the definitions of our object terms must look syntactically like values. (Alternatively, we could have used the rank-2 record types of OCaml to maintain the necessary polymorphism.) Thus, we represent an object expression in OCaml as a functor from `Symantics` to an appropriate semantic domain. This is essentially the same as the constraint `Symantics repr =>` in the Haskell embedding.

2.2 Two tagless interpreters

Having abstracted our term representation over the interpreter, we are now ready to present a series of interpreters. Each interpreter is an instance of the `Symantics` class in Haskell and a module implementing the `Symantics` signature in MetaOCaml.

The first interpreter evaluates an object term to its value in the metalanguage. The module below interprets each object-language operation as the corresponding metalanguage operation.

```
module R = struct type ('c,'dv) repr = 'dv (* no wrappers *)
  let int  (x:int)  = x        let bool (b:bool) = b
  let lam  f        = f        let app  e1 e2    = e1 e2
  let fix  f        = let rec self n = f self n in self
  let add  e1 e2    = e1 + e2  let mul  e1 e2    = e1 * e2
  let leq  x y      = x <= y
  let if_  eb et ee = if eb then et () else ee () end
```

As in §1.2, this interpreter is patently tagless, using neither a universal type nor any pattern matching: the operation `add` is really OCaml's addition, and `app` is OCaml's application. To run our examples, we instantiate the `EX` functor from §2.1 with `R`: `module EXR = EX(R)`. Thus, `EXR.test1 ()` evaluates to the untagged boolean value `true`. It is obvious to the compiler that pattern matching cannot fail, because there is no pattern matching. Evaluation can only fail to yield a value due to interpreting `fix`. (The source code shows a total interpreter `L` that measures the size of each object term.) We can also generalize from `R` to all interpreters; these propositions follow immediately from the soundness of the metalanguage's type system.

Proposition 2. *If an object term e encoded in the metalanguage has type t, then evaluating e in the interpreter `R` either continues indefinitely or terminates with a value of the same type t.*

Proposition 3. *If an implementation of `Symantics` never gets stuck, then the type system of the object language is sound with respect to the dynamic semantics defined by that implementation.*

3 A tagless compiler (or, a staged interpreter)

Besides immediate evaluation, we can compile our object language into OCaml code using MetaOCaml's staging facilities. MetaOCaml represents future-stage expressions of type t as values of type `('c,` t`) code`, where `'c` is the environment classifier [6, 34]. Code values are created by a *bracket* form `.<`e`>.`, which quotes the expression e for evaluation at a future stage. The *escape* `.~`e must occur within a bracket and specifies that the expression e must be evaluated at the current stage; its result, which must be a code value, is spliced into the code being built by the enclosing bracket. The *run* form `.!`e evaluates the future-stage code value e by compiling and linking it at run time. Bracket, escape, and run are akin to quasi-quotation, unquotation, and `eval` of Lisp.

Inserting brackets and escapes appropriately into the evaluator `R` above yields the simple compiler `C` in Fig. 5(a). This is a straightforward staging of `module R`. This compiler produces unoptimized code. For example, interpreting our `test1` with Fig. 5(b) gives the code value `.<(fun x_6 -> x_6) true>.` of inferred type `('c, bool) C.repr`. Interpreting `testpowfix7` with Fig. 5(c) gives a code value with many apparent β- and η-redexes, Fig. 5(d). This compiler does not incur

```
(a) module C = struct type ('c,'dv) repr = ('c,'dv) code
      let int (x:int)  = .<x>.            let bool (b:bool) = .<b>.
      let lam f        = .<fun x -> .~(f .<x>.)>.
      let app e1 e2    = .<.~e1 .~e2>.
      let fix f =  .<let rec self n = .~(f .<self>.) n in self>.
      let add e1 e2    = .<.~e1 + .~e2>.  let mul e1 e2 = .<.~e1 * .~e2>.
      let leq x y      = .<.~x <= .~y>.
      let if_ eb et ee = .<if .~eb then .~(et ()) else .~(ee ())>. end
(b) let module E = EX(C) in E.test1 ()
(c) let module E = EX(C) in E.testpowfix7
(d) .<fun x_1 -> (fun x_2 -> let rec self_3 = fun n_4 ->
     (fun x_5 -> if x_5 <= 0 then 1 else x_2 * self_3 (x_5 + (-1)))
     n_4 in self_3) x_1 7>.
```

Fig. 5. The tagless staged interpreter `C`

any interpretive overhead: the code produced for $\lambda x.\, x$ is simply `fun x_6 -> x_6`. The resulting code obviously contains no tags and no pattern matching. The environment classifiers here, like the tuple types in §1.2, make it a type error to run an open expression. The accompanying code shows the Haskell implementation.

4 A tagless partial evaluator

Surprisingly, we can write a partial evaluator using the idea above, namely to build object terms using ordinary functions rather than data constructors. We present this partial evaluator in a sequence of three attempts. It uses no universal type and no tags for object types. We then discuss residualization and binding-time analysis. Our partial evaluator is a modular extension of the evaluator in §2.2 and the compiler in §3, in that it uses the former to reduce static terms and the latter to build dynamic terms.

4.1 Avoiding polymorphic lift

Roughly, a partial evaluator interprets each object term to yield either a static (present-stage) term (using `R`) or a dynamic (future-stage) term (using `C`). To distinguish between static and dynamic terms, we might try to define `repr` in the partial evaluator as `type ('c,'dv) repr = S0 of ('c,'dv) R.repr | E0 of ('c,'dv) C.repr`. Integer and boolean literals are immediate, present-stage values. Addition yields a static term (using `R.add`) if and only if both operands are static; otherwise we extract the dynamic terms from the operands and add them using `C.add`. We use `C.int` to convert from the static term `('c,int) R.repr`, which is just `int`, to the dynamic term.

Whereas `mul` and `leq` are as easy to define as `add`, we encounter a problem with `if_`. Suppose that the first argument to `if_` is a dynamic term (of type `('c,bool) C.repr`), the second a static term (of type `('c,'a) R.repr`), and the third a dynamic term (of type `('c,'a) C.repr`). We then need to convert the static term to dynamic, but there is no polymorphic "lift" function, of type `'a -> ('c,'a) C.repr`, to send a value to the future stage [34, 37].

Our `Symantics` only includes separate lifting methods `bool` and `int`, not a parametrically polymorphic lifting method, for good reason: When compiling to a first-order target language such as machine code, booleans, integers, and functions may well be represented differently. Thus, compiling polymorphic lift requires intensional type analysis. To avoid needing polymorphic lift, we turn to Asai's technique [1, 32]: build a dynamic term alongside every static term.

4.2 Delaying binding-time analysis

We switch to the data type `type ('c,'dv) repr = P1 of ('c,'dv) R.repr option * ('c,'dv) C.repr` so that a partially evaluated term always contains a dynamic component and sometimes contains a static component. By distributivity, the two alternative constructors of an `option` value, `Some` and `None`, tag each partially evaluated term with a phase: either present or future. This tag is not an object type tag: all pattern matching below is exhaustive. Because the future-stage component is always present, we can now define the polymorphic function `let abstr1 (P1 (_,dyn)) = dyn` of type `('c,'dv) repr -> ('c,'dv) C.repr` to extract it without requiring polymorphic lift into `C`. We then try to define the interpreter `P1`—and get as far as the first-order constructs of our object language, including `if_`.

```
module P1 : Symantics = struct
  let int (x:int) = P1 (Some (R.int x), C.int x)
  let add e1 e2 = match (e1,e2) with
    | (P1 (Some n1,_),P1 (Some n2,_)) -> int (R.add n1 n2)
    | _ -> P1 (None,(C.add (abstr1 e1) (abstr1 e2)))
  let if_ = function
    | P1 (Some s,_) -> fun et ee -> if s then et () else ee ()
    | eb -> fun et ee -> P1 (None, C.if_ (abstr1 eb)
                                  (fun () -> abstr1 (et ()))
                                  (fun () -> abstr1 (ee ())))
```

However, we stumble on functions. According to our definition of `P1`, a partially evaluated object function, such as the identity $\lambda x.\, x$ embedded in OCaml as `lam (fun x -> x) : ('c,'a->'a) P1.repr`, consists of a dynamic part (type `('c,'a->'a) C.repr`) and maybe a static part (type `('c,'a->'a) R.repr`). The dynamic part is useful when this function is passed to another function that is only dynamically known, as in $\lambda k.\, k(\lambda x.\, x)$. The static part is useful when this function is applied to a static argument, as in $(\lambda x.\, x)$ true. Neither part, however, lets us *partially* evaluate the function, that is, compute as much as possible statically when it is applied to a mix of static and dynamic inputs. For example, the partial evaluator should turn $\lambda n.\, (\lambda x.\, x)n$ into $\lambda n.\, n$ by substituting n for x in the body of $\lambda x.\, x$ even though n is not statically known. The same static function, applied to different static arguments, can give both static and dynamic results: we want to simplify $(\lambda y.\, x \times y)0$ to 0 but $(\lambda y.\, x \times y)1$ to x.

To enable these simplifications, we delay binding-time analysis for a static function until it is applied, that is, until `lam f` appears as the argument of `app`. To do so, we have to incorporate `f` as it is into the `P1.repr` data structure: the representation for a function type `'a->'b` should be one of

```
S1 of ('c,'a) repr -> ('c,'b) repr | E1 of ('c,'a->'b) C.repr
P1 of (('c,'a) repr -> ('c,'b) repr) option * ('c,'a->'b) C.repr
```

unlike P1.repr of int or bool. That is, we need a nonparametric data type, something akin to type-indexed functions and type-indexed types, which Oliveira and Gibbons [22] dub the *typecase* design pattern. Thus, typed partial evaluation, like typed CPS transformation, inductively defines a map from source types to target types that performs case distinction on the source type. In Haskell, typecase can be equivalently implemented either with GADTs or with type-class functional dependencies [22]. The accompanying code shows both approaches, neither portable to OCaml. In addition, the problem of nonexhaustive pattern-matching reappears in the GADT approach because GHC 6.6.1 cannot see that a particular type of a GADT value precludes certain constructors. Thus GADTs fail to make it *syntactically* apparent that pattern matching is exhaustive.

4.3 The "final" solution

Let us re-examine the problem in §4.2. What we would ideally like is to write type ('c,'dv) repr = P1 of (repr_pe ('c,'dv)) R.repr option * ('c,'dv) C.repr where repr_pe is the type function defined by

```
repr_pe ('c,int) = ('c,int); repr_pe ('c,bool) = ('c,bool)
repr_pe ('c,'a->'b) = ('c,'a) repr -> ('c,'b) repr
```

Although we can use type classes to define this type function in Haskell, that is not portable to MetaOCaml. However, these three typecase alternatives are already present in existing methods of Symantics. A simple and portable solution thus emerges: we bake repr_pe into the signature Symantics. We recall from Figure 4 in §2.1 that the repr type constructor took two arguments 'c and 'dv. We add an argument 'sv for the result of applying repr_pe to 'dv. Figure 6 shows the new signature.

```
module type Symantics = sig type ('c,'sv,'dv) repr
  val int : int -> ('c,int,int) repr
  val lam : (('c,'sa,'da) repr -> ('c,'sb,'db) repr as 'x)
            -> ('c,'x,'da -> 'db) repr
  val app : ('c,'x,'da -> 'db) repr
            -> (('c,'sa,'da) repr -> ('c,'sb,'db) repr as 'x)
  val fix : ('x -> 'x) -> (('c, ('c,'sa,'da) repr -> ('c,'sb,'db) repr,
                            'da -> 'db) repr as 'x)
  val add : ('c,int,int) repr -> ('c,int,int) repr -> ('c,int,int) repr
  val if_ : ('c,bool,bool) repr
            -> (unit->'x) -> (unit->'x) -> (('c,'sa,'da) repr as 'x) end
```

Fig. 6. A (Meta)OCaml embedding of our object language that supports partial evaluation (bool, mul, leq are elided)

The interpreters R, L and C above only use the old type arguments 'c and 'dv, which are treated by the new signature in the same way. Hence, all that needs to change in these interpreters to match the new signature is to add a phantom type argument 'sv to repr. For example, the compiler C now begins module C = struct type ('c,'sv,'dv) repr = ('c,'dv) code with the rest the same. In contrast, the partial evaluator P relies on the type argument 'sv.

Figure 7 shows the partial evaluator P. Its type repr literally expresses the type equation for repr_pe above. The function abstr extracts a future-stage code value from the result of partial evaluation. Conversely, the function pdyn injects a code value into the repr type. As in §4.2, we build dynamic terms alongside any static ones to avoid polymorphic lift.

```
module P = struct
  type ('c,'sv,'dv) repr = {st: 'sv option; dy: ('c,'dv) code}
  let abstr {dy = x} = x    let pdyn x = {st = None; dy = x}

  let int  (x:int)  = {st = Some (R.int x);  dy = C.int x}
  let add e1 e2 = match e1, e2 with
  | {st = Some 0}, e | e, {st = Some 0} -> e
  | {st = Some m}, {st = Some n} -> int (R.add m n)
  | _ -> pdyn (C.add (abstr e1) (abstr e2))
  let if_ eb et ee = match eb with
  | {st = Some b} -> if b then et () else ee ()
  | _ -> pdyn (C.if_ (abstr eb) (fun () -> abstr (et ()))
                               (fun () -> abstr (ee ())))
  let lam f = {st = Some f; dy = C.lam (fun x -> abstr (f (pdyn x)))}
  let app ef ea = match ef with {st = Some f} -> f ea
                | _ -> pdyn (C.app (abstr ef) (abstr ea)) end
```

Fig. 7. Our partial evaluator (bool, mul, leq and fix are elided)

The static portion of the interpretation of lam f is Some f, which just wraps the HOAS function f. The interpretation of app ef ea checks to see if ef is such a wrapped HOAS function. If it is, we apply f to the concrete argument ea, giving us a chance to perform static computations (see the example below). If ef has only a dynamic part, we residualize.

To illustrate how to add optimizations, we improve add (and mul, elided) to simplify the generated code using the monoid (and ring) structure of int: not only is addition performed statically (using R) when both operands are statically known, but it is eliminated when one operand is statically 0; similarly for multiplication by 0 or 1. Such optimizations can be quite effective in a large language with more base types and primitive operations.

Any partial evaluator must decide how much to unfold recursion. Our code naïvely unfolds fix whenever the argument is static. In the accompanying source code is a conservative alternative P.fix that unfolds recursion only once, then residualizes. Many sophisticated approaches have been developed to decide how much to unfold [17], but this issue is orthogonal to our presentation.

Given this implementation of P, our running example let module E = EX(P) in E.test1 () evaluates to {P.st = Some true; P.dy = .<true>.} of type ('a, bool, bool) P.repr. Unlike with C in §3, a β-reduction has been statically performed to yield true. More interestingly, whereas testpowfix7 compiles to a code value with many β-redexes in §3, the partial evaluation let module E = EX(P) in E.testpowfix7 gives the desired result

```
{P.st = Some <fun>;
 P.dy = .<fun x -> x * (x * (x * (x * (x * (x * x)))))>.}
```

All pattern-matching in P is *syntactically* exhaustive, so it is patent to the meta-language implementation that P never gets stuck. Further, all pattern-matching occurs during partial evaluation, only to check if a value is known statically,

never what type it has. In other words, our partial evaluator tags phases (with `Some` and `None`) but not object types.

5 Related work

Our initial motivation came from several papers [23, 24, 33, 37] that use embedded interpreters to justify advanced type systems, in particular GADTs. We admire all this technical machinery, but these motivating examples do not need it. Although GADTs may indeed be simpler and more flexible, they are unavailable in mainstream ML, and their implementation in GHC 6.6.1 fails to detect exhaustive pattern matching. We also wanted to find the minimal set of widespread language features needed for tagless type-preserving interpretation.

Even a simply typed λ-calculus obviously supports self-interpretation, provided we use universal types [33]. The ensuing tagging overhead motivated Taha et al. [33] to propose tag elimination, which however does not statically guarantee that all tags will be removed [23].

Pašalić et al. [23], Taha et al. [33], Xi et al. [37], and Peyton Jones et al. [24] seem to argue as follows that a self-interpreter of a typed language cannot be tagless or Jones-optimal: (1) One needs to encode a typed language in a typed language based on a sum type (at some level of the hierarchy); (2) A *direct* interpreter for such an encoding of a typed language in a typed language requires either advanced types or tagging overhead; (3) Thus, an indirect interpreter is necessary, which needs a universal type and hence tagging. While the logic is sound, we (following Yang [38]) showed that the first step's premise is not valid.

Danvy and López [8] discuss Jones optimality at length and apply HOAS to typed self-interpretation. However, their source language is untyped. Therefore, their object-term encoding has tags, and their interpreter can raise run-time errors. Nevertheless, HOAS lets the partial evaluator remove all the tags. In contrast, our object encoding and interpreters do not have tags to start with and obviously cannot raise run-time errors.

Our partial evaluator establishes a bijection `repr_pe` between static and dynamic types (the valid values of `'sv` and `'dv`), and between static and dynamic terms. It is customary to implement such a bijection using an injection-projection pair, as done for interpreters [4, 27], partial evaluation [7], and type-level functions [22]. As explained in §4.3, we avoid injection and projection at the type level by adding an argument to `repr`. Our solution could have been even more straightforward if MetaOCaml provided total type-level functions such as `repr_pe` in §4.3—simple type-level computations ought to become mainstream.

At the term level, we also avoid converting between static and dynamic terms by building them in parallel, using Asai's method [1]. This method type-checks in Hindley-Milner once we deforest the object term representation. Put another way, we manual apply type-level partial evaluation to our type functions (see §4.3) to obtain simpler types acceptable to MetaOCaml.

Sumii and Kobayashi [32] also use Asai's method, to combine online and offline partial evaluation. They predate us in deforesting the object term representation to enable tagless partial evaluation. We strive for modularity by reusing interpreters for individual stages [31]: our partial evaluator `P` reuses our tagless evaluator `R` and tagless compiler `C`, so it is patent that the *output* of `P` never gets stuck. It would be interesting to try to derive a *cogen* [35] in the same manner.

It is common to implement an embedded DSL by providing multiple interpretations of host-language pervasives such as addition and application. It is also common to use phantom types to rule out ill-typed object terms, as done in Lava [5] and by Rhiger [29]. However, these approaches are not tagless because they still use universal types, such as Lava's `Bit` and `NumSig`, and Rhiger's `Raw` (his Fig. 2.2) and `Term` (his Chap. 3), which incur the attendant overhead of pattern matching. The universal type also greatly complicates the soundness and completeness proofs of embedding [29], whereas our proofs are trivial. Rhiger's approach does not support typed CPS transformation (his §3.3.4).

We are not the first to implement a typed interpreter for a typed language. Läufer and Odersky [18] use type classes to implement a metacircular interpreter (rather than a self-interpreter) of a typed version of the SK language, which is quite different from our object language. Their interpreter appears to be tagless, but they could not have implemented a compiler or partial evaluator in the same way, since they rely heavily on injection-projection pairs.

Fiore [10] and Balat et al. [3] also build a tagless partial evaluator, using delimited control operators. It is type-directed, so the user must represent, as a term, the type of every term to be partially evaluated. We shift this work to the type checker of the metalanguage. By avoiding term-level type representations, our approach makes it easier to perform algebraic simplifications (as in §4.3).

We encode terms in elimination form, as a coalgebraic structure. Pfenning and Lee [26] first described this basic idea and applied it to metacircular interpretation. Our approach, however, can be implemented in mainstream ML and supports type inference, typed CPS transformation and partial evaluation. In contrast, Pfenning and Lee conclude that partial evaluation and program transformations "do not seem to be expressible" even using their extension to F_ω, perhaps because their avoidance of general recursive types compels them to include the polymorphic lift that we avoid in §4.1.

Our encoding of the type function `repr_pe` in §4.3 emulates type-indexed types and is related to intensional type analysis [13, 14]. However, our object language and running examples in HOAS include `fix`, which intensional type analysis cannot handle [37]. Our final approach seems related to Washburn and Weirich's approach to HOAS using catamorphisms and anamorphisms [36].

We could not find work that establishes that the *typed* λ-calculus has a final coalgebra structure. (See Honsell and Lenisa [15] for the untyped case.)

We observe that higher-rank and higher-kind polymorphism lets us type-check and compile object terms separately from interpreters. This is consistent with the role of polymorphism in the separate compilation of modules [30].

6 Conclusions

We solve the problem of embedding a typed object language in a typed metalanguage without using GADTs, dependent types, or a universal type. Our family of interpreters include an evaluator, a compiler, a partial evaluator, and CPS transformers. It is patent that they never get stuck, because we represent object

types as metalanguage types. This work makes it safer and more efficient to embed DSLs in practical metalanguages such as Haskell and ML.

Our main idea is to represent object programs not in an initial algebra but using the existing coalgebraic structure of the λ-calculus. More generally, to squeeze more invariants out of a type system as simple as Hindley-Milner, we shift the burden of representation and computation from consumers to producers: encoding object terms as calls to metalanguage functions (§1.2); build dynamic terms alongside static ones (§4.1); simulating type functions for partial evaluation (§4.3) and CPS transformation. This shift also underlies fusion, functionalization, and amortized complexity analysis.

Our representation of object terms in elimination form encodes primitive recursive folds over the terms. We still have to understand if and how non-primitively recursive operations can be supported.

References

[1] Asai, Kenichi. 2001. Binding-time analysis for both static and dynamic expressions. *New Generation Computing* 20(1):27–52.

[2] Baars, Arthur I., and S. Doaitse Swierstra. 2002. Typing dynamic typing. In *ICFP*, 157–166.

[3] Balat, Vincent, Roberto Di Cosmo, and Marcelo P. Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, 64–76.

[4] Benton, P. Nick. 2005. Embedded interpreters. *JFP* 15(4):503–542.

[5] Bjesse, Per, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware design in Haskell. In *ICFP*, 174–184.

[6] Calcagno, Cristiano, Eugenio Moggi, and Walid Taha. 2004. ML-like inference for classifiers. In *ESOP*, 79–93.

[7] Danvy, Olivier. 1996. Type-directed partial evaluation. In *POPL*, 242–257.

[8] Danvy, Olivier, and Pablo E. Martínez López. 2003. Tagging, encoding, and Jones optimality. In *ESOP*, 335–347.

[9] Davies, Rowan, and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48(3):555–604.

[10] Fiore, Marcelo P. 2002. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *PPDP*, 26–37.

[11] Fogarty, Seth, Emir Pasalic, Jeremy Siek, and Walid Taha. 2007. Concoqtion: Indexed types now! In *PEPM*.

[12] Glück, Robert. 2002. Jones optimality, binding-time improvements, and the strength of program specializers. In *ASIA-PEPM*, 9–19.

[13] Harper, Robert, and J. Gregory Morrisett. 1995. Compiling polymorphism using intensional type analysis. In *POPL*, 130–141.

[14] Hinze, Ralf, Johan Jeuring, and Andres Löh. 2004. Type-indexed data types. *Sci. Comput. Program.* 51(1-2):117–151.

[15] Honsell, Furio, and Marina Lenisa. 1999. Coinductive characterizations of applicative structures. *Math. Structures in Comp. Sci.* 9(4):403–435.

[16] Hudak, Paul. 1996. Building domain-specific embedded languages. *ACM Comp. Surv.* 28(4es):196.

[17] Jones, Neil D., Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice-Hall.

[18] Läufer, Konstantin, and Martin Odersky. 1993. Self-interpretation and reflection in a statically typed language. In *OOPSLA/ECOOP workshop on object-oriented reflection and metalevel architectures.*

[19] MetaOCaml. `http://www.metaocaml.org`.

[20] Miller, Dale, and Gopalan Nadathur. 1987. A logic programming approach to manipulating formulas and programs. In *IEEE symp. on logic programming*, 379–388.

[21] Nanevski, Aleksandar, and Frank Pfenning. 2005. Staged computation with names and necessity. *JFP* 15(6):893–939.

[22] Oliveira, Bruno César dos Santos, and Jeremy Gibbons. 2005. TypeCase: A design pattern for type-indexed functions. In *Haskell workshop*, 98–109.

[23] Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *ICFP*, 157–166.

[24] Peyton Jones, Simon L., Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ICFP*, 50–61.

[25] Pfenning, Frank, and Conal Elliott. 1988. Higher-order abstract syntax. In *PLDI*, 199–208.

[26] Pfenning, Frank, and Peter Lee. 1991. Metacircularity in the polymorphic λ-calculus. *Theor. Comp. Sci.* 89(1):137–159.

[27] Ramsey, Norman. 2005. ML module mania: A type-safe, separately compiled, extensible interpreter. In *ML workshop.*

[28] Reynolds, John C. 1972. Definitional interpreters for higher-order programming languages. In *Proc. ACM Natl. Conf.*, vol. 2, 717–740. Repr. with a foreword in *HOSC* 11(4):363–397.

[29] Rhiger, Morten. 2001. Higher-Order program generation. Ph.D. thesis, BRICS, Denmark.

[30] Shao, Zhong. 1998. Typed cross-module compilation. In *ICFP*, 141–152.

[31] Sperber, Michael, and Peter Thiemann. 1997. Two for the price of one: Composing partial evaluation and compilation. In *PLDI*, 215–225.

[32] Sumii, Eijiro, and Naoki Kobayashi. 2001. A hybrid approach to online and offline partial evaluation. *HOSC* 14(2–3):101–142.

[33] Taha, Walid, Henning Makholm, and John Hughes. 2001. Tag elimination and Jones-optimality. In *PADO*, 257–275. LNCS 2053.

[34] Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL*, 26–37.

[35] Thiemann, Peter. 1996. Cogen in six lines. In *ICFP*, 180–189.

[36] Washburn, Geoffrey, and Stephanie Weirich. 2003. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ICFP*, 249–262.

[37] Xi, Hongwei, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL*, 224–235.

[38] Yang, Zhe. 1998. Encoding types in ML-like languages. In *ICFP*, 289–300.

A Gentle Introduction to Multi-stage Programming*

Walid Taha

Department of Computer Science, Rice University, Houston, TX, USA
taha@rice.edu

Abstract. Multi-stage programming (MSP) is a paradigm for developing generic software that does not pay a runtime penalty for this generality. This is achieved through concise, carefully-designed language extensions that support runtime code generation and program execution. Additionally, type systems for MSP languages are designed to statically ensure that dynamically generated programs are type-safe, and therefore require no type checking after they are generated.
This hands-on tutorial is aimed at the reader interested in learning the basics of MSP practice. The tutorial uses a freely available MSP extension of OCaml called MetaOCaml, and presents a detailed analysis of the issues that arise in staging an interpreter for a small programming language. The tutorial concludes with pointers to various resources that can be used to probe further into related topics.

1 Introduction

Although program generation has been shown to improve code reuse, product reliability and maintainability, performance and resource utilization, and developer productivity, there is little support for *writing* generators in mainstream languages such as C or Java. Yet a host of basic problems inherent in program generation can be addressed effectively by a programming language designed specifically to support writing generators.

1.1 Problems in Building Program Generators

One of the simplest approaches to writing program generators is to represent the program fragments we want to generate as either strings or data types ("abstract syntax trees"). Unfortunately, both representations have disadvantages. With the string encoding, we represent the code fragment `f (x,y)` simply as `"f (x,y)"`. Constructing and combining fragments represented by strings can be done concisely. But there is no *automatically verifiable* guarantee that programs constructed in this manner are syntactically correct. For example, `"f (,y)"` can have the static type `string`, but this string is clearly *not* a syntactically correct program.

* Supported by NSF ITR-0113569 and NSF CCR-0205542.

With the data type encoding the situation is improved, but the best we can do is ensure that any generated program is syntactically correct. We cannot use data types to ensure that generated programs are well-typed. The reason is that data types can represent context-free sets accurately, but usually not context sensitive sets. Type systems generally define context sensitive sets (of programs). Constructing data type values that represent trees can be a bit more verbose, but a quasi-quotation mechanism [1] can alleviate this problem and make the notation as concise as that of strings.

In contrast to the strings encoding, MSP languages statically ensure that any generator only produces syntactically well-formed programs. Additionally, statically typed MSP languages statically ensure that any generated program is also well-typed.

Finally, with both string and data type representations, ensuring that there are no name clashes or inadvertent variable captures *in the generated program* is the responsibility of the programmer. This is essentially the same problem that one encounters with the C macro system. MSP languages ensure that such inadvertent capture is not possible. We will return to this issue when we have seen one example of MSP.

1.2 The Three Basic MSP Constructs

We can illustrate how MSP addresses the above problems using MetaOCaml [2], an MSP extension of OCaml [9]. In addition to providing traditional imperative, object-oriented, and functional constructs, MetaOCaml provides three constructs for staging. The constructs are called Brackets, Escape, and Run. Using these constructs, the programmer can change the order of evaluation of terms. This capability can be used to reduce the overall cost of a computation.

Brackets (written `.<...>.`) can be inserted around any expression to delay its execution. MetaOCaml implements delayed expressions by dynamically generating source code at runtime. While using the source code representation is not the only way of implementing MSP languages, it is the simplest. The following short interactive MetaOCaml session illustrates the behavior of Brackets[1]:

```
# let a = 1+2;;
val a : int = 3
# let a = .<1+2>.;;
val a : int code = .<1+2>.
```

Lines that start with `#` are what is entered by the user, and the following line(s) are what is printed back by the system. Without the Brackets around `1+2`, the addition is performed right away. With the Brackets, the result is a piece of code representing the program `1+2`. This code fragment can either be used as part of another, bigger program, or it can be compiled and executed.

[1] Some versions of MetaOCaml developed after December 2003 support environment classifiers [21]. For these systems, the type `int code` is printed as `('a,int) code`. To follow the examples in this tutorial, the extra parameter `'a` can be ignored.

In addition to delaying the computation, Brackets are also reflected in the type. The type in the last declaration is `int code`. The type of a code fragment reflects the type of the value that such code should produce when it is executed. Statically determining the type of the generated code allows us to avoid writing generators that produce code that cannot be typed. The code type constructor distinguishes delayed values from other values and prevents the user from accidentally attempting unsafe operations (such as `1 + .<5>.`).

Escape (written `.~...`) allows the combination of smaller delayed values to construct larger ones. This combination is achieved by "splicing-in" the argument of the Escape in the context of the surrounding Brackets:

```
# let b = .<.~a * .~a >. ;;
val b : int code = .<(1 + 2) * (1 + 2)>.
```

This declaration binds `b` to a new delayed computation `(1+2)*(1+2)`.

Run (written `.!...`) allows us to compile and execute the dynamically generated code without going outside the language:

```
# let c = .! b;;
val c : int = 9
```

Having these three constructs as part of the programming language makes it possible to use runtime code generation and compilation as part of any library subroutine. In addition to not having to worry about generating temporary files, static type systems for MSP languages can assure us that no runtime errors will occur in these subroutines (c.f. [17]). Not only can these type systems exclude generation-time errors, but they can also ensure that generated programs are both syntactically well-formed and well-typed. Thus, the ability to statically type-check the safety of a computation is not lost by staging.

1.3 Basic Notions of Equivalence

As an aside, there are two basic equivalences that hold for the three MSP constructs [18]:

$$\texttt{.\~{}}\ \texttt{.<}e\texttt{>.} = e$$
$$\texttt{.!}\ \texttt{.<}e\texttt{>.} = e$$

Here, a value v can include usual values such as integers, booleans, and lambdas, as well as Bracketed terms of the form `.<`e`>.`. In the presentation above, we use e for an expression where all Escapes are enclosed by a matching set of Brackets. The rules for Escape and Run are identical. The distinction between the two constructs is in the notion of values: the expression in the value `.<`e`>.` cannot contain Escapes that are not locally surrounded by their own Brackets. An expression e is unconstrained as to where Run can occur.

Avoiding accidental name capture. Consider the following staged function:

```
# let rec h n z = if n=0 then z
                  else .<(fun x -> .~(h (n-1) .<x+ .~z>.)) n>.;;
val h : int -> int code -> int code = <fun>
```

If we erase the annotations (to get the "unstaged" version of this function) and apply it to `3` and `1`, we get `7` as the answer. If we apply the staged function above to `3` and `.<1>.`, we get the following term:

```
.<(fun x_1 -> (fun x_2 -> (fun x_3 -> x_3 + (x_2 + (x_1 + 1))) 1) 2) 3>.
```

Whereas the source code only had `fun x -> ...` inside Brackets, this code fragment was generated three times, and each time it produced a different `fun x_i -> ...` where `i` is a different number each time. If we run the generated code above, we get `7` as the answer. We view it as a highly desirable property that the results generated by staged programs are related to the results generated by the unstaged program. The reader can verify for herself that if the `x`s were not renamed and we allowed variable capture, the answer of running the staged program would be different from `7`. Thus, automatic renaming of bound variables is not so much a feature; rather, it is the absence of renaming that seems like a bug.

1.4 Organization of This Paper

The goal of this tutorial is to familiarize the reader with the basics of MSP. To this end, we present a detailed example of how MSP can be used to build staged interpreters. The practical appeal of staged interpreters lies in that they can be almost as simple as interpreter-based language implementations and at the same time be as efficient as compiler-based ones. To this end, Section 2 briefly describes an idealized method for developing staged programs in a programming language that provides MSP constructs. The method captures a process that a programmer iteratively applies while developing a staged program. This method will be used repeatedly in examples in the rest of the paper. Section 3 constitutes the technical payload of the paper, and presents a series of more sophisticated interpreters and staged interpreters for a toy programming language. This section serves both to introduce key issues that arise in the development of a staged interpreter (such as error handling and binding-time improvements), and to present a realistic example of a series of iterative refinements that arise in developing a satisfactory staged interpreter. Section 4 concludes with a brief overview of additional resources for learning more about MSP.

2 How Do We Write MSP Programs?

Good abstraction mechanisms can help the programmer write more concise and maintainable programs. But if these abstraction mechanisms degrade performance, they will not be used. The primary goal of MSP is to help the programmer reduce the runtime overhead of sophisticated abstraction mechanisms. Our

hope is that having such support will allow programmers to write higher-level and more reusable programs.

2.1 A Basic Method for Building MSP Programs

An idealized method for building MSP programs can be described as follows:

1. A single-stage program is developed, implemented, and tested.
2. The organization and data-structures of the program are studied to ensure that they can be used in a staged manner. This analysis may indicate a need for "factoring" some parts of the program and its data structures. This step can be critical step toward effective MSP. Fortunately, it has been thoroughly investigated in the context of partial evaluation where it is known as *binding-time engineering* [7].
3. Staging annotations are introduced to explicitly specify the evaluation order of the various computations of the program. The staged program may then be tested to ensure that it achieves the desired performance.

The method described above, called *multi-stage programming with explicit annotations [22]*, can be summarized by the slogan:

A Staged Program = A Conventional Program + Staging Annotations

We consider the method above to be idealized because, in practice, it is useful to iterate through steps 1-3 in order to determine the program that is most suitable for staging. This iterative process will be illustrated in the rest of this paper. But first, we consider a simpler example where the method can be applied directly.

2.2 A Classic Example

A common source of performance overhead in generic programs is the presence of parameters that do not change very often, but nevertheless cause our programs to repeatedly perform the work associated with these inputs. To illustrate, we consider the following classic function in MetaOCaml: [7]

```
let rec power (n, x) =
  match n with
    0 -> 1 | n -> x * (power (n-1, x));;
```

This function is generic in that it can be used to compute `x` raised to *any* nonnegative exponent `n`. While it is convenient to use generic functions like `power`, we often find that we have to pay a price for their generality. Developing a good understanding of the source of this performance penalty is important, because it is exactly what MSP will help us eliminate. For example, if we need to compute the second power often, it is convenient to define a special function:

```
let power2 (x) = power (2,x);;
```

In a functional language, we can also write it as follows:

```
let power2 = fun x -> power (2,x);;
```

Here, we have taken away the formal parameter `x` from the left-hand side of the equality and replaced it by the equivalent "`fun x ->`" on the right-hand side. To use the function `power2`, all we have to do is to apply it as follows:

```
let answer = power2 (3);;
```

The result of this computation is `9`. But notice that every time we apply `power2` to some value `x` it calls the `power` function with parameters `(2,x)`. And even though the first argument will always be `2`, evaluating `power (2,x)` will always involve calling the function recursively two times. This is an undesirable overhead, because we *know* that the result can be more efficiently computed by multiplying `x` by itself. Using only unfolding and the definition of `power`, we know that the answer can be computed more efficiently by:

```
let power2 = fun x -> 1*x*x;;
```

We also do not want to write this by hand, as there may be many other specialized power functions that we wish to use. So, can we *automatically* build such a program?

In an MSP language such as MetaOCaml, all we need to do is to *stage* the power function by annotating it:

```
let rec power (n, x) =
  match n with
    0 -> .<1>. | n -> .<.~x * .~(power (n-1, x))>.;;
```

This function still takes two arguments. The second argument is no longer an integer, but rather, a *code of type integer*. The return type is also changed. Instead of returning an integer, this function will return a code of type integer. To match this return type, we insert Brackets around `1` in the first branch on the third line. By inserting Brackets around the multiplication expression, we now return a code of integer instead of just an integer. The Escape around the recursive call to `power` means that it is performed immediately.

The staging constructs can be viewed as "annotations" on the original program, and are fairly unobtrusive. Also, we are able to type check the code both outside and inside the Brackets in essentially the same way that we did before. If we were using strings instead of Brackets, we would have to sacrifice static type checking of delayed computations.

After annotating `power`, we have to annotate the uses of `power`. The declaration of `power2` is annotated as follows:

```
let power2 = .! .<fun x -> .~(power (2,.<x>.))>.;;
```

Evaluating the application of the Run construct will compile and if execute its argument. Notice that this declaration is essentially the same as what we used to define `power2` before, except for the staging annotations. The annotations say that we wish to construct the code for a function that takes one argument (`fun x ->`). We also do not spell out what the function itself should do; rather , we use the Escape construct (`.~`) to make a call to the staged `power`. The result

of evaluating this declaration behaves exactly as if we had defined it "by hand" to be `fun x -> 1*x*x`. Thus, it will behave exactly like the first declaration of `power2`, but it will run as though we had written the specialized function by hand. The staging constructs allowed us to eliminate the runtime overhead of using the generic power function.

3 Implementing DSLs Using Staged Interpreters

An important application for MSP is the implementation of domain-specific languages (DSLs) [4]. Languages can be implemented in a variety of ways. For example, a language can be implemented by an interpreter or by a compiler. Compiled implementations are often orders of magnitude faster than interpreted ones, but compilers have traditionally required significant expertise and took orders of magnitude more time to implement than interpreters. Using the method outlined in Section 2, we can start by first writing an interpreter for a language, and then stage it to get *a staged interpreter*. Such staged interpreters can be as simple as the interpreter we started with and at the same time have performance comparable to that of a compiler. This is possible because the staged interpreter becomes effectively a *translator* from the DSL to the host language (in this case OCaml). Then, by using MetaOCaml's Run construct, the total effect is in fact a function that takes DSL programs and produces machine code[2].

To illustrate this approach, this section investigates the implementation of a toy programming language that we call `lint`. This language supports integer arithmetic, conditionals, and recursive functions. A series of increasingly more sophisticated interpreters and staged interpreters are used to give the reader a hands-on introduction to MSP. The complete code for the implementations described in this section is available online [10].

3.1 Syntax

The syntax of expressions in this language can be represented in OCaml using the following data type:

```
type exp = Int of int | Var of string | App of string * exp
         | Add of exp * exp | Sub of exp * exp
         | Mul of exp * exp | Div of exp * exp | Ifz of exp * exp * exp

type def = Declaration of string * string * exp
type prog = Program of def list * exp
```

Using the above data types, a small program that defines the factorial function and then applies it to 10 can be concisely represented as follows:

[2] If we are using the MetaOCaml native code compiler. If we are using the bytecode compiler, the composition produces bytecode.

```
Program ([Declaration
          ("fact","x", Ifz(Var "x",
                     Int 1,
                     Mul(Var"x",
                        (App ("fact", Sub(Var "x",Int 1))))))
         ],
         App ("fact", Int 10))
```

OCaml lex and yacc can be used to build parsers that take textual representations of such programs and produce abstract syntax trees such as the above. In the rest of this section, we focus on what happens after such an abstract syntax tree has been generated.

3.2 Environments

To associate variable and function names with their values, an interpreter for this language will need a notion of an environment. Such an environment can be conveniently implemented as a function from names to values. If we look up a variable and it is not in the environment, we will raise an exception (let's call it `Yikes`). If we want to extend the environment (which is just a function) with an association from the name `x` to a value `v`, we simply return a new environment (a function) which first tests to see if its argument is the same as `x`. If so, it returns `v`. Otherwise, it looks up its argument in the original environment. All we need to implement such environments is the following:

```
exception Yikes

(* env0, fenv : 'a -> 'b *)

let env0 = fun x -> raise Yikes         let fenv0 = env0

(* ext : ('a -> 'b) -> 'a -> 'b -> 'a -> 'b *)

let ext env x v = fun y -> if x=y then v else env y
```

The types of all three functions are polymorphic. Type variables such as `'a` and `'b` are implicitly universally quantified. This means that they can be later instantiated to more specific types. Polymorphism allows us, for example, to define the initial function environment `fenv0` as being exactly the same as the initial variable environment `env0`. It will also allow us to use the same function `ext` to extend both kinds of environments, even when their types are instantiated to the more specific types such as:

```
env0 : string -> int                    fenv0 : string -> (int -> int)
```

3.3 A Simple Interpreter

Given an environment binding variable names to their runtime values and function names to values (these will be the formal parameters `env` and `fenv`, respectively), an interpreter for an expression `e` can be defined as follows:

```
(* eval1 : exp -> (string -> int) -> (string -> int -> int) -> int *)

let rec eval1 e env fenv =
match e with
  Int i -> i
| Var s -> env s
| App (s,e2) -> (fenv s)(eval1 e2 env fenv)
| Add (e1,e2) -> (eval1 e1 env fenv)+(eval1 e2 env fenv)
| Sub (e1,e2) -> (eval1 e1 env fenv)-(eval1 e2 env fenv)
| Mul (e1,e2) -> (eval1 e1 env fenv)*(eval1 e2 env fenv)
| Div (e1,e2) -> (eval1 e1 env fenv)/(eval1 e2 env fenv)
| Ifz (e1,e2,e3) -> if (eval1 e1 env fenv)=0
                    then (eval1 e2 env fenv)
                    else (eval1 e3 env fenv)
```

This interpreter can now be used to define the interpreter for declarations and programs as follows:

```
(* peval1 : prog -> (string -> int) -> (string -> int -> int) -> int *)

let rec peval1 p env fenv=
    match p with
     Program ([],e) -> eval1 e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        let rec f x = eval1 e1 (ext env s2 x) (ext fenv s1 f)
        in peval1 (Program(tl,e)) env (ext fenv s1 f)
```

In this function, when the list of declarations is empty (`[]`), we simply use `eval1` to evaluate the body of the program. Otherwise, we recursively interpret the list of declarations. Note that we also use `eval1` to interpret the body of function declarations. It is also instructive to note the three places where we use the environment extension function `ext` on both variable and function environments.

The above interpreter is a complete and concise specification of what programs in this language should produce when they are executed. Additionally, this style of writing interpreters follows quite closely what is called the denotational style of specifying semantics, which can be used to specify a wide range of programming languages. It is reasonable to expect that a software engineer can develop such implementations in a short amount of time.

Problem: The cost of abstraction. If we evaluate the factorial example given above, we will find that it runs about 20 times slower than if we had written this example directly in OCaml. The main reason for this is that the interpreter repeatedly traverses the abstract syntax tree during evaluation. Additionally, environment lookups in our implementation are not constant-time.

3.4 The Simple Interpreter Staged

MSP allows us to keep the conciseness and clarity of the implementation given above and also eliminate the performance overhead that traditionally we would have had to pay for using such an implementation. The overhead is avoided by staging the above function as follows:

```
(* eval2 : exp -> (string -> int code) -> (string -> (int -> int) code)
              -> int code *)

let rec eval2 e env fenv =
match e with
  Int i -> .<i>.
| Var s -> env s
| App (s,e2) -> .<.~(fenv s).~(eval2 e2 env fenv)>.
...
| Div (e1,e2)-> .<.~(eval2 e1 env fenv)/ .~(eval2 e2 env fenv)>.
| Ifz (e1,e2,e3) -> .<if .~(eval2 e1 env fenv)=0
                      then .~(eval2 e2 env fenv)
                      else .~(eval2 e3 env fenv)>.

(* peval2 : prog -> (string -> int code) -> (string -> (int -> int) code)
                -> int code *)

let rec peval2 p env fenv=
    match p with
     Program ([],e) -> eval2 e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
     .<let rec f x = .~(eval2 e1 (ext env s2 .<x>.)
                                 (ext fenv s1 .<f>.))
       in .~(peval2 (Program(tl,e)) env (ext fenv s1 .<f>.))>.
```

If we apply `peval2` to the abstract syntax tree of the factorial example (given above) and the empty environments `env0` and `fenv0`, we get back the following code fragment:

```
.<let rec f = fun x -> if x = 0 then 1 else x * (f (x - 1)) in (f 10)>.
```

This is exactly the same code that we would have written by hand for that specific program. Running this program has exactly the same performance as if we had written the program directly in OCaml.

The staged interpreter is a function that takes abstract syntax trees and produces MetaOCaml programs. This gives rise to a simple but often overlooked fact [19,20]:

A staged interpreter is a translator.

It is important to keep in mind that the above example is quite simplistic, and that further research is need to show that we can apply MSP to realistic programming languages. Characterizing what MSP cannot do is difficult, because of the need for technical expressivity arguments. We take the more practical approach of explaining what MSP can do, and hope that this gives the reader a working understanding of the scope of this technology.

3.5 Error Handling

Returning to our example language, we will now show how direct staging of an interpreter does not always yield satisfactory results. Then, we will give an example of the wide range of techniques that can be used in such situations.

A generally desirable feature of programming languages is error handling. The original implementation of the interpreter uses the division operation, which can raise a divide-by-zero exception. If the interpreter is part of a bigger system, we probably want to have finer control over error handling. In this case, we would modify our original interpreter to perform a check before a division, and return a special value `None` if the division could not be performed. Regular values that used to be simply `v` will now be represented by `Some v`. To do this, we will use the following standard datatype:

```
type 'a option = None | Just of 'a;;
```

Now, such values will have to be propagated and dealt with everywhere in the interpreter:

```
(* eval3 : exp -> (string -> int) -> (string -> int -> int option)
               -> int option *)

let rec eval3 e env fenv =
match e with
  Int i -> Some i
| Var s -> Some (env s)
| App (s,e2) -> (match (eval3 e2 env fenv) with
                   Some x -> (fenv s) x
                 | None   -> None)
| Add (e1,e2) -> (match (eval3 e1 env fenv, eval3 e2 env fenv)
                   with (Some x, Some y) -> Some (x+y)
                       | _ -> None) ...
| Div (e1,e2) -> (match (eval3 e1 env fenv, eval3 e2 env fenv)
                   with (Some x, Some y) ->
                          if y=0 then None
                                 else Some (x/y)
                      | _ -> None)
| Ifz (e1,e2,e3) -> (match (eval3 e1 env fenv) with
                      Some x -> if x=0 then (eval3 e2 env fenv)
                                       else (eval3 e3 env fenv)
                    | None   -> None)
```

Compared to the unstaged interpreter, the performance overhead of adding such checks is marginal. But what we really care about is the staged setting. Staging `eval3` yields:

```
(* eval4 : exp -> (string -> int code)
               -> (string -> (int -> int option) code)
               -> (int option) code *)

let rec eval4 e env fenv =
match e with
  Int i -> .<Some i>.
| Var s -> .<Some .~(env s)>.
| App (s,e2) -> .<(match .~(eval4 e2 env fenv) with
                    Some x -> .~(fenv s) x
                  | None   -> None)>.
| Add (e1,e2) -> .<(match (.~(eval4 e1 env fenv),
                           .~(eval4 e2 env fenv)) with
                     (Some x, Some y) -> Some (x+y)
                   | _ -> None)>. ...
| Div (e1,e2) -> .<(match (.~(eval4 e1 env fenv),
                           .~(eval4 e2 env fenv)) with
                     (Some x, Some y) ->
                       if y=0 then None
                              else Some (x/y)
                   | _ -> None)>.
| Ifz (e1,e2,e3) -> .<(match .~(eval4 e1 env fenv) with
                        Some x -> if x=0 then
                                    .~(eval4 e2 env fenv)
                                  else
                                    .~(eval4 e3 env fenv)
                      | None   -> None)>.
```

Problem: The cost of error handling. Unfortunately, the performance of code generated by this staged interpreter is typically 4 times slower than the first staged interpreter that had no error handling. The source of the runtime cost becomes apparent when we look at the generated code:

```
.<let rec f =
   fun x ->
    (match (Some (x)) with
       Some (x) ->
        if (x = 0) then (Some (1))
        else
         (match
            ((Some (x)),
             (match
                (match ((Some (x)), (Some (1))) with
                   (Some (x), Some (y)) ->
                    (Some ((x - y)))
                 | _ -> (None)) with
                Some (x) -> (f x)
              | None -> (None))) with
            (Some (x), Some (y)) ->
             (Some ((x * y)))
          | _ -> (None))
     | None -> (None)) in
  (match (Some (10)) with
     Some (x) -> (f x)
   | None -> (None))>.
```

The generated code is doing much more work than before, because at every operation we are checking to see if the values we are operating with are proper values or not. Which branch we take in every `match` is determined by the explicit form of the value being matched.

Half-solutions. One solution to such a problem is certainly adding a pre-processing analysis. But if we can avoid generating such inefficient code in the first place it would save the time wasted both in generating these unnecessary checks and in performing the analysis. More importantly, with an analysis, we may never be certain that all unnecessary computation is eliminated from the generated code.

3.6 Binding-time Improvements

The source of the problem is the `if` statement that appears in the interpretation of `Div`. In particular, because `y` is bound inside Brackets, we cannot perform the test `y=0` while we are building for these Brackets. As a result, we cannot immediately determine if the function should return a `None` or a `Some` value. This affects the type of the whole staged interpreter, and effects the way we interpret all programs even if they do not contain a use of the `Div` construct.

The problem can be avoided by what is called a *binding-time improvement* in the partial evaluation literature [7]. It is essentially a transformation of the program that we are staging. The goal of this transformation is to allow better staging. In the case of the above example, one effective binding time improvement is to rewrite the interpreter in continuation-passing style (CPS) [5], which produces the following code:

```
(* eval5 : exp -> (string -> int) -> (string -> int -> int)
              -> (int option -> 'a) -> 'a *)

let rec eval5 e env fenv k =
match e with
  Int i -> k (Some i)
| Var s -> k (Some (env s))
| App (s,e2) -> eval5 e2 env fenv
                 (fun r -> match r with
                   Some x -> k (Some ((fenv s) x))
                 | None   -> k None)
| Add (e1,e2) -> eval5 e1 env fenv
                  (fun r ->
                    eval5 e2 env fenv
                      (fun s -> match (r,s) with
                        (Some x, Some y) -> k (Some (x+y))
                       | _ -> k None)) ...
| Div (e1,e2) -> eval5 e1 env fenv
                  (fun r ->
                    eval5 e2 env fenv
                      (fun s -> match (r,s) with
                        (Some x, Some y) ->
                          if y=0 then k None
                                 else k (Some (x/y))
                       | _ -> k None)) ...

(* pevalK5 : prog -> (string -> int) -> (string -> int -> int)
                  -> (int option -> int) -> int *)

let rec pevalK5 p env fenv k =
    match p with
     Program ([],e) -> eval5 e env fenv k
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        let rec f x = eval5 e1 (ext env s2 x) (ext fenv s1 f) k
          in pevalK5 (Program(tl,e)) env (ext fenv s1 f) k

exception Div_by_zero;;

(* peval5 : prog -> (string -> int) -> (string -> int -> int) -> int *)

let peval5 p env fenv =
     pevalK5 p env fenv (function Some x -> x
                              | None -> raise Div_by_zero)
```

In the unstaged setting, we can use the CPS implementation to get the same functionality as the direct-style implementation. But note that the two algorithms are *not the same*. For example, performance of the CPS interpreter is in fact worse than the previous one. But when we try to stage the new interpreter, we find that we can do something that we could not do in the direct-style interpreter. In particular, the CPS interpreter can be staged as follows:

```
(* eval6 : exp -> (string -> int code) -> (string -> (int -> int) code)
               -> (int code option -> 'b code) -> 'b code *)

let rec eval6 e env fenv k =
match e with
  Int i -> k (Some .<i>.)
| Var s -> k (Some (env s))
| App (s,e2) -> eval6 e2 env fenv
                 (fun r -> match r with
                   Some x -> k (Some .<.~(fenv s) .~x>.)
                 | None   -> k None)
| Add (e1,e2) -> eval6 e1 env fenv
                  (fun r ->
                    eval6 e2 env fenv
                      (fun s -> match (r,s) with
                        (Some x, Some y) ->
                          k (Some .<.~x + .~y>.)
                       | _ -> k None)) ...
| Div (e1,e2) -> eval6 e1 env fenv
```

```
                    (fun r ->
                      eval6 e2 env fenv
                        (fun s -> match (r,s) with
                          (Some x, Some y) ->
                            .<if .~y=0 then .~(k None)
                              else .~(k (Some .<.~x / .~y>.))>.
                         | _ -> k None))
| Ifz (e1,e2,e3) -> eval6 e1 env fenv
                     (fun r -> match r with
                       Some x -> .<if .~x=0 then
                                      .~(eval6 e2 env fenv k)
                                   else
                                      .~(eval6 e3 env fenv k)>.
                      | None   -> k None)

(* peval6 : prog -> (string -> int code) -> (string -> (int -> int) code)
                 -> int code *)

let peval6 p env fenv =
     pevalK6 p env fenv (function Some x -> x
                                | None -> .<raise Div_by_zero>.)
```

The improvement can be seen at the level of the type of `eval6`: the `option` type occurs outside the `code` type, which suggests that it can be eliminated in the first stage. What we could not do before is to Escape the application of the continuation to the branches of the `if` statement in the `Div` case. The extra advantage that we have when staging a CPS program is that we are applying the continuation multiple times, which is essential for performing the computation in the branches of an `if` statement. In the unstaged CPS interpreter, the continuation is always applied exactly once. Note that this is the case even in the `if` statement used to interpret `Div`: The continuation does *occur* twice (once in each branch), but only one branch is ever taken when we evaluate this statement. But in the the staged interpreter, the continuation is indeed duplicated and applied multiple times.[3]

The staged CPS interpreter generates code that is exactly the same as what we got from the first interpreter as long as the program we are interpreting does not use the division operation. When we use division operations (say, if we replace the code in the body of the fact example with `fact (20/2)`) we get the following code:

```
.<let rec f =
   fun x -> if (x = 0) then 1 else (x * (f (x - 1)))
  in if (2 = 0) then (raise (Div_by_zero)) else (f (20 / 2))>.
```

[3] This means that converting a program into CPS can have disadvantages (such as a computationally expensive first stage, and possibly code duplication). Thus, CPS conversion must be used judiciously [8].

3.7 Controlled Inlining

So far we have focused on eliminating unnecessary work from generated programs. Can we do better? One way in which MSP can help us do better is by allowing us to unfold function declarations for a fixed number of times. This is easy to incorporate into the first staged interpreter as follows:

```
(* eval7 : exp -> (string -> int code)
               -> (string -> int code -> int code) -> int code *)

let rec eval7 e env fenv =
match e with
... most cases the same as eval2, except
| App (s,e2) -> fenv s (eval7 e2 env fenv)

(* repeat : int -> ('a -> 'a) -> 'a -> 'a *)

let rec repeat n f =
  if n=0 then f else fun x -> f (repeat (n-1) f x)

(* peval7 : prog -> (string -> int code)
                 -> (string -> int code -> int code) -> int code *)

let rec peval7 p env fenv=
    match p with
     Program ([],e) -> eval7 e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        .<let rec f x =
            .~(let body cf x =
                    eval7 e1 (ext env s2 x) (ext fenv s1 cf) in
                 repeat 1 body (fun y -> .<f .~y>.) .<x>.)
          in .~(peval7 (Program(tl,e)) env
                      (ext fenv s1 (fun y -> .<f .~y>.)))>.
```

The code generated for the factorial example is as follows:

```
.<let rec f =
   fun x ->
    if (x = 0) then 1
    else
     (x*(if ((x-1)=0) then 1 else (x-1)*(f ((x-1)-1)))))
  in (f 10)>.
```

This code can be expected to be faster than that produced by the first staged interpreter, because only one function call is needed for every two iterations.

Avoiding Code Duplication The last interpreter also points out an important issue that the multi-stage programmer must pay attention to: code duplication. Notice that the term `x-1` occurs three times in the generated code. In the result of the first staged interpreter, the subtraction only occurred once. The duplication

of this term is a result of the inlining that we perform on the body of the function. If the argument to a recursive call was even bigger, then code duplication would have a more dramatic effect both on the time needed to compile the program and the time needed to run it.

A simple solution to this problem comes from the partial evaluation community: we can generate `let` statements that replace the expression about to be duplicated by a simple variable. This is only a small change to the staged interpreter presented above:

```
let rec eval8 e env fenv =
match e with
... same as eval7 except for
| App (s,e2) -> .<let x= .~(eval8 e2 env fenv)
                 in .~(fenv s .<x>.)>. ...
```

Unfortunately, in the current implementation of MetaOCaml this change does not lead to a performance improvement. The most likely reason is that the bytecode compiler does not seem to perform `let`-floating. In the native code compiler for MetaOCaml (currently under development) we expect this change to be an improvement.

Finally, both error handling and inlining can be combined into the same implementation:

```
(* eval9: exp -> (string -> int code) -> (string -> int code -> int code)
              -> (int code option -> 'b code) -> 'b code *)

let rec eval9 e env fenv k =
match e with
... same as eval6, except
| App (s,e2) -> eval9 e2 env fenv
                 (fun r -> match r with
                   Some x -> k (Some ((fenv s) x))
                 | None   -> k None)

(* pevalK9 : prog -> (string -> int code)
                 -> (string -> int code -> int code)
                  -> (int code option -> int code) -> int code  *)

let rec pevalK9 p env fenv k =
    match p with
     Program ([],e) -> eval9 e env fenv k
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        .<let rec f x =
          .~(let body cf x =
                 eval9 e1 (ext env s2 x) (ext fenv s1 cf) k in
             repeat 1 body (fun y -> .<f .~y>.) .<x>.)
          in .~(pevalK9 (Program(tl,e)) env
                       (ext fenv s1 (fun y -> .<f .~y>.)) k)>.
```

3.8 Measuring Performance in MetaOCaml

MetaOCaml provides support for collecting performance data, so that we can empirically verify that staging does yield the expected performance improvements. This is done using three simple functions. The first function must be called before we start collecting timings. It is called as follows:

```
Trx.init_times ();;
```

The second function is invoked when we want to gather timings. Here we call it twice on both `power2 (3)` and `power2'(3)` where `power2'` is the staged version:

```
Trx.timenew "Normal" (fun () ->(power2  (3)));;
Trx.timenew "Staged" (fun () ->(power2' (3)));;
```

Each call to `Trx.timenew` causes the argument passed last to be run as many times as this system needs to gather a reliable timing. The quoted strings simply provide hints that will be printed when we decide to print the summary of the timings. The third function prints a summary of timings:

```
Trx.print_times ();;
```

The following table summarizes timings for the various functions considered in the previous section:[4]

Program	Description of Interpreter	Fact10	Fib20
(none)	OCaml implementations	100%	100%
eval1	Simple	1,570%	1,736%
eval2	Simple staged	100%	100%
eval3	Error handling (EH)	1,903%	2,138%
eval4	EH staged	417%	482%
eval5	CPS, EH	2,470%	2,814%
eval6	CPS, EH, staged	100%	100%
eval7	Inlining, staged	87%	85%
eval8	Inlining, no duplication, staged	97%	97%
eval9	Inlining, CPS, EH, staged	90%	85%

Timings are normalized relative to handwritten OCaml versions of the DSL programs that we are interpreting (`Fact10` and `Fib20`). Lower percentages mean faster execution. This kind of normalization does not seem to be commonly used in the literature, but we believe that it has an important benefit. In particular, it serves to emphasize the fact that the performance of the resulting specialized programs should really be compared to specialized hand-written programs. The fact that `eval9` is more than 20 times faster than `eval3` is often irrelevant to a programmer who rightly considers `eval3` to be an impractically inefficient implementation. As we mentioned earlier in this paper, the goal of MSP is to remove unnecessary runtime costs associated with abstractions, and identifying the right reference point is essential for knowing when we have succeeded.

[4] System specifications: MetaOCaml bytecode interpreter for OCaml 3.07+2 running under Cygwin on a Pentium III machine, Mobile CPU, 1133MHz clock, 175 MHz bus, 640 MB RAM. Numbers vary when Linux is used, and when the native code compiler is used.

4 To Probe Further

The sequence of interpreters presented here is intended to illustrate the iterative nature of the process of exploring how to stage an interpreter for a DSL so that it can be turned into a satisfactory translator, which when composed with the Run construct of MetaOCaml, gives a compiler for the DSL.

A simplifying feature of the language studied here is that it is a first-order language with only one type, `int`. For richer languages we must define a datatype for values, and interpreters will return values of this datatype. Using such a datatype has an associated runtime overhead. Eliminating the overhead for such datatypes can be achieved using either a post-processing transformation called tag elimination [20] or MSP languages with richer static type systems [14]. MetaOCaml supports an experimental implementation of tag elimination [2].

The extent to which MSP is effective is often dictated by the surrounding environment in which an algorithm, program, or system is to be used. There are three important examples of situations where staging can be beneficial:

- We want to minimize total cost of all stages *for most inputs*. This model applies, for example, to implementations of programming languages. The cost of a simple compilation followed by execution is *usually* lower than the cost of interpretation. For example, the program being executed usually contains loops, which typically incur large overhead in an interpreted implementation.
- We want to minimize *a weighted average* of the cost of all stages. The weights reflect the relative frequency at which the result of a stage can be reused. This situation is relevant in many applications of symbolic computation. Often, solving a problem symbolically, and then graphing the solution at a thousand points can be cheaper than numerically solving the problem a thousand times. This cost model can make a symbolic approach worthwhile even when it is 100 times more expensive than a direct numerical one. (By symbolic computation we simply mean computation where free variables are values that will only become available at a later stage.)
- We want to minimize the cost of the *last stage*. Consider an embedded system where the *sin* function may be implemented as a large look-up table. The cost of constructing the table is not relevant. Only the cost of computing the function at run-time is. The same applies to optimizing compilers, which may spend an unusual amount of time to generate a high-performance computational library. The cost of optimization is often not relevant to the users of such libraries.

The last model seems to be the most commonly referenced one in the literature, and is often described as "there is ample time between the arrival of different inputs", "there is a significant difference between the frequency at which the various inputs to a program change", and "the performance of the program matters only after the arrival of its last input".

Detailed examples of MSP can be found in the literature, including term-rewriting systems [17] and graph algorithms [12]. The most relevant example to domain-specific program generation is on implementing a small imperative language [16]. The present tutorial should serve as good preparation for approaching the latter example.

An implicit goal of this tutorial is to prepare the reader to approach introductions to partial evaluation [3] and partial evaluation of interpreters in particular [6]. While these two works can be read without reading this tutorial, developing a full appreciation of the ideas they discuss requires either an understanding of the internals of partial evaluators or a basic grasp of multi-stage computation. This tutorial tries to provide the latter.

Multi-stage languages are both a special case of meta-programming languages and a generalization of multi-level and two-level languages. Taha [17] gives definitions and a basic classification for these notions. Sheard [15] gives a recent account of accomplishments and challenges in meta-programming. While multi-stage languages are "only" a special case of meta-programming languages, their specialized nature has its advantages. For example, it is possible to formally prove that they admit strong algebraic properties even in the untyped setting, but this is not the case for more general notions of meta-programming [18]. Additionally, significant advances have been made over the last six years in static typing for these languages (c.f. [21]). It will be interesting to see if such results can be attained for more general notions of meta-programming.

Finally, the MetaOCaml web site will be continually updated with new research results and academic materials relating to MSP [11].

Acknowledgments: Stephan Jorg Ellner, John Garvin, Christoph Hermann, Samah Mahmeed, and Kedar Swadi read and gave valuable comments on this tutorial. Thanks are also due to the reviewers, whose detailed comments have greatly improved this work, as well as to the four editors of this volume, who have done an extraordinary job organizing the Dagstuhl meeting and organizing the preparation of this volume.

References

1. Alan Bawden. Quasiquotation in LISP. In O. Danvy, editor, *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, San Antonio, 1999. University of Aarhus, Dept. of Computer Science. Invited talk.
2. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
3. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.
4. Krzysztof Czarnecki1, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In this volume.
5. O. Danvy. Semantics-directed compilation of non-linear patterns. Technical Report 303, Indiana University, Bloomington, Indiana, USA, 1990.

6. Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.
7. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
8. J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *1994 ACM Conference on Lisp and Functional Programming, Orlando, Florida, June 1994*, pages 227–238. New York: ACM, 1994.
9. Xavier Leroy. Objective Caml, 2000. Available from `http://caml.inria.fr/ocaml/`.
10. Complete source code for `lint`. Available online from `http://www.metaocaml.org/examples/lint.ml`, 2003.
11. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2003.
12. The MetaML Home Page, 2000. Provides source code and documentation online at `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.
13. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
14. Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.
15. Tim Sheard. Accomplishments and research challenges in meta-programming. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineer SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 2–44. ACM, Springer, October 2002.
16. Tim Sheard, Zine El-Abidine Benaissa, and Emir Pašalić. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, 1999. USENIX.
17. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [13].
18. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)*, Boston, 2000. ACM Press.
19. Walid Taha and Henning Makholm. Tag elimination – or – type specialisation is a type-indexed effect. In Subtyping and Dependent Types in Programming, APPSEM Workshop. INRIA technical report, 2000.
20. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinksi, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 257–275, 2001.
21. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
22. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.

Interpreting quotations*

Chung-chieh Shan

Draft of 2008-03-21 06:53:12

Mixed quotes are quotes that appear to mix mention and use, or direct and indirect quotation, such as (1).

(1) Quine says that quotation ‘has a certain anomalous feature.’ (Davidson 1979)

This paper argues that mixed quotation is a general phenomenon that pervades the interpretation of language. In fact, most of our speech consists of mixed quotes of ourselves and each other. Of course, because the vast majority of utterances do not call for quotation marks in print, this point is only plausible if we broaden the notion of mixed quotation to include many of them. My first order of business is thus to explain a broader notion of mixed quotation. I then use this notion to analyze naming and quantification.

1. The essence of mixed quotation

Informally speaking, I take a mixed quote to mean what someone uses the quoted expression to mean (Geurts and Maier 2003). For example, (1) means that Quine says that quotation has the property that Quine uses ‘has a certain anomalous feature’ to mean. The quoted expression need not be grammatical, as (2) shows.

(2) The president said he has an ‘ecelectic’ reading list. (Maier 2007)

This sentence means that the president said he has a reading list with the property that he uses ‘ecelectic’ to mean.

In a sense, mixed quotation internalizes the semantic interpretation function—the familiar double square brackets ⟦ ⟧ for denotation—and relativizes it to a speaker, be it Quine or the president. Of course, the use of a mixed quote often accomplishes far more than presupposing that someone uses the quoted expression to mean something and denoting that meaning. For example, the speakers of (2) and (3) can distance

*Thanks to Mark Baker, Chris Barker, Sam Cumming, David Dowty, Pauline Jacobson, Michael Johnson, Oleg Kiselyov, Eli Bohmer Lebow, Ernie Lepore, Emar Maier, Mats Rooth, Ken Safir, Roger Schwarzschild, Stuart Shieber, Matthew Stone, Dylan Thurston, and the Rutgers linguistics department. This is work in progress; please send comments to ccshan@rutgers.edu.

themselves from the speakers they quote, perhaps by triggering the implicature that they would not themselves use the quoted expressions in the same ways.

(3) I am sorry to have used an ‘epithet’.

Nevertheless, I focus here on the idea that a mixed quote means what someone uses the quoted expression to mean, and only promise to treat the other accomplishments of a mixed quote as different ways to use it (Cappelen and Lepore 2003).

My formal treatment of this idea is motivated by two empirical facts: first, mixed quotes can be nested; second, they can apply to constructions, not just expressions.

1.1. Nested mixed quotes

Mixed quotation can be nested (iterated), just as pure quotation can be.

(4) The politician said she is ‘sorry to have used an ‘epithet’ ’.

On one reading, (4) means that the politician said she has the property that she uses the phrase ‘sorry to have used an ‘epithet’ ’ to mean. Presumably—that is, assuming that the politician is a normal English speaker—this property is to be sorry to have used an element of the set that someone unspecified uses the word ‘epithet’ to mean. The outer quotation level in (4) distinguishes the speaker’s sense of ‘sorry’ from the politician’s; the inner level distinguishes the politician’s sense of ‘epithet’ from others’. The outer level may even contain only the inner level, as in (5).

(5) The politician said she is sorry to have used an ‘ ‘epithet’ ’.

The fact that one mixed quote can contain another already follows from the fact that a mixed quote can contain any form, not necessarily a grammatical expression in any language. Yet it is worth noting that the speaker of the outer quote presumably uses the inner quote to mean what someone uses the inner-quoted expression to mean, because this presumption lets us explain why the meaning of (4) involves what someone uses the word ‘epithet’ to mean.

1.2. Mixed quotes of constructions

A construction can be quoted, just as an ordinary expression can be.

(6) The politician admitted that she ‘lied [her] way into [her job]’.

(7) It is a long story how I lied my way into this despicable position of deception.

(8) lied my way into this despicable position of deception

Thanks to the square brackets in (6) or their spoken counterpart, the sentence is true if the politician said (7) as a normal English speaker. More precisely, the sentence

is true just in case the politician admitted that she the property $g\,y\,z$, where g is the ternary relation that she used the construction 'lied ... way into ...' to mean, y is her, and z is her job. Intuitively, what is quoted in (6) is not an expression such as (8), but a construction that combines the subexpressions 'my way' and 'this despicable position of deception' to form (8). The same construction also combines the meanings of the subexpressions to form the meaning of (8).

An ordinary expression can be treated as a special case of a construction, namely a nullary one—a construction that takes no input. The binary construction quoted in the example above is a canonical non-nullary construction, but less canonical ones can be mixed-quoted as well, subject to the practical and pragmatic difficulty of punctuating and intoning the quotes so as to convey the speaker's intention.

(9) John doesn't know much French, but he thinks he does and tries to show it off whenever possible. At dinner the other day, he ordered not '[some dessert] à la mode' but 'à la mode [some dessert]'.

On one reading, (9) can be true if John ordered using the words 'à la mode apple pie' but not 'apple pie à la mode'. That is, the second mixed quote in (9) is of a unary construction. The form of the construction maps expressions to expressions: it puts 'à la mode' before a dessert name. The meaning of the construction maps desserts to desserts. What John actually ordered was of course not an expression but the result of applying to some dessert the meaning map that John used the construction to mean.

At least some mixed quotes of non-constituents can be better analyzed as mixed quotes of constructions.

(10) Mary allowed as how her dog ate 'odd things, when left to his own devices'. (Abbott 2003)

The mixed-quoted non-constituent in (10) includes not just the noun phrase 'odd things' and the adverbial phrase 'when left to his own devices' but also Mary's juxtaposing them, indicated in writing by quoting a comma. Maier's quote-breaking analysis (2007) decomposes the quoted non-constituent into two phrases and thus overlooks their juxtaposition. Instead, we can treat the mixed quote as one of a unary construction that builds a verb phrase from a transitive verb, say the verb phrase 'ate odd things, when left to his own devices' from the transitive verb 'ate', or the verb phrase 'devoured odd things, when left to his own devices' from the transitive verb 'devoured'. One attempt at notating such an analysis is (11).

(11) Mary allowed as how her dog '[ate] odd things, when left to his own devices'.

(12) Fido devoured odd things, when left to his own devices.

(13) Whereas under human supervision Fido ate odd things, when left to his own devices he would only eat Nutrapup.

On this analysis, (10) holds if Mary says (12), but not if Mary only says (13).

1.3. Distinguishing syntactic and semantic interjection

Conventional punctuation using square brackets fails to distinguish between two ways to interrupt a quote and interject words used from the perspective of the quoter. The first way, exemplified above, is for the meaning of the interjected words to combine *semantically* with the (rest of the) quote: in (6), the meaning of 'her' and 'her job', say the politician and her job, may serve as arguments to some functional meaning of the construction 'lied ... way into ...'. The second way, exemplified below, is for the meaning of the interjected words to combine *syntactically* with the (rest of the) quote.

(14) The secret guide suggested that interested eaters 'kiss up to [name redacted], class of 2008, for a good meal' at the Ivy.

(15) Randal L. Schwartz writes: 'Something like this, perhaps? [some typical R. Schwartz code, short, simple, clear, efficient, etc....]'

The secret guide in (14) did not suggest that interested eaters kiss up to a name; it used, not mentioned, a redacted name. Schwartz in (15) actually wrote some typical code, not (just) referred to it.

To avoid this ambiguity of square brackets, we notate semantic interjection %[like this] and syntactic interjection ∼[like this]. We further distinguish mixed quotes from pure and direct quotes by notating mixed quotes !⌈like this⌉ and pure and direct quotes ⌈like this⌉. For example, we notate (14) and (15) as follows.

(16) The secret guide suggested that interested eaters !⌈kiss up to ∼[name redacted], class of 2008, for a good meal⌉ at the Ivy.

(17) Randal L. Schwartz writes: ⌈Something like this, perhaps? ∼[some typical R. Schwartz code, short, simple, clear, efficient, etc....]⌉

This notation follows that of *multistage programming languages* (Taha 2004) and suggests that mixed quotation internalizes semantic interpretation. It becomes crucial when we discuss quantification in §2.2.

1.4. A formal model of grammatical constructions

Mixed quotes of mixed quotes and of constructions motivate the following model of grammatical constructions in which to discuss the form and meaning of mixed quotation.

Fix a set X of syntactic objects (forms) and a set Y of semantic objects (meanings). An n-ary *construction* is an ordered pair $\langle f, g\rangle$ where f is a partial function from X^n to X and g is a partial function from Y^n to Y. We can *apply* the construction $\langle f, g\rangle$ to the *constituents* $\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle$, each a form-meaning pair, to build the form-meaning pair $\langle f\,x_1 \ldots x_n, g\,y_1 \ldots y_n\rangle$, as long as f is defined at $x_1 \ldots x_n$ and g is defined at $y_1 \ldots y_n$. For clarity, we sometimes write $x_{1\ldots n}$ instead of $x_1 \ldots x_n$.

These definitions leave it wide open what forms and meanings are. A fan of Heim and Kratzer (1998) might take each lexical entry to be a nullary construction pairing a phrase structure with a denotation, Merge and function application to be a binary construction, and Move and predicate abstraction to be a unary construction. A fan of Steedman (1996) might take each lexical entry to be a nullary construction pairing a categorized string with a denotation, concatenation and function application to be a binary construction, and each type-shifting operation to be a unary construction. A construction may be an inference rule in a logic or a step in the execution of a computer program. Even the odd linguist (see §2.2) who pines for 'meta-constructions'—constructions over constructions—can express them by treating constructions as expressions and meta-constructions (along with the closure conditions below) as constructions. These details do not concern us, in large part thanks to the conditions imposed by the following definition.

A *grammar* R is a set of constructions that satisfies two closure conditions.

Identity The pair of identity functions $\langle \lambda x.\, x, \lambda y.\, y\rangle$ is in R as a unary construction.

Composition If $\langle f, g\rangle$ is an $(n+1)$-ary construction in R, and if $\langle f', g'\rangle$ is an n'-ary construction in R, then the $(n+n')$-ary construction

$$\begin{aligned}\langle \lambda x_{1\dots i-1} x'_{1\dots n'} x_{i+1\dots n+1}.\, f\, x_{1\dots i-1}(f' x'_{1\dots n'}) x_{i+1\dots n+1},\\ \lambda y_{1\dots i-1} y'_{1\dots n'} y_{i+1\dots n+1}.\, g\, y_{1\dots i-1}(g' y'_{1\dots n'}) y_{i+1\dots n+1} \rangle\end{aligned}$$

is also in R, for $i = 1, \dots, n+1$. If we identify expressions with nullary constructions, then application is a special case of composition.

This definition is inspired by *operads without permutation* (May 1997).

The grammar *generated* by a set of constructions S is the smallest grammar containing S. The closure conditions let us treat mixed quotes of *primitive* constructions (those in S) and *derived* constructions (those in R but not S) alike. For example, any reasonable grammar of English in this model probably contains a nullary construction whose form component is 'Mary saw John'. Once one speaker uses this construction, another may then quote it—be it the composition of a binary construction '– saw –' with the nullary constructions 'Mary' and 'John' or of a whole bunch of Merge, Move, and lexical constructions.

We can now be a bit more specific about what it means for a speaker to use a form to mean something, or to use a construction. We assume that the use of language pairs forms with meanings, be it subjectively in an idiolect or objectively in a language game. For example, Alice in her use of language might pair the form 'Mary saw John' with the meaning that Mary saw John, and the president in his use of language might pair the form 'I have an ecelectic reading list' with the meaning that he (de se) has an eclectic reading list. When a speaker s pairs the form x with the meaning y, we say that s uses x to mean y, or that s uses the nullary construction $\langle x, y\rangle$.

But that is not all. A speaker often justifies the use of a construction by applying the closure conditions above to other constructions. For example, our model English speaker Alice justifies the nullary construction 'Mary saw John' (here we notate a construction by its form) by composing some binary construction '– saw –' with the nullary constructions 'Mary' and 'John'. In fact, one construction use may be justified in multiple *equivalent* ways: Alice justifies 'Mary saw John' by composing 'Mary' with '– saw John' as much as she does by composing 'Mary saw –' with 'John' (Barker 2007). If s uses a construction and justifies it by applying the closure conditions to other constructions, then s also uses those other constructions. That is, if we draw the justification of a used construction as a parse or proof tree whose nodes are primitive constructions, then any connected part of the tree depicts a used construction as well.

(18)

– saw –

Mary John

1.5. Mixed quotes, formally

I analyze mixed quotes as constructions of the form

(19) $$\langle\ Qf,\ \iota g.\, x \text{ uses the construction } \langle f, g\rangle\ \rangle.$$

Here f and Qf are two partial functions from X^n to X, related in some systematic way Q yet to be specified, so the set X of forms needs to express the intonation or punctuation of quotation. The meaning component of this construction is anaphoric to some discourse referent x and presupposes that the speaker x uses f to mean a partial function g from Y^n to Y. These anaphoric and presuppositional dependencies are part of the construction's meaning and remain to be resolved—either semantically as the meaning is composed with other meanings, or pragmatically as the resulting utterance is used in discourse. On this analysis, then, the set Y of meanings needs to express these dependencies, and mixed quotes are *not* constructions of the form

(20) $$\langle\ Qf,\ g\ \rangle$$

where some speaker x uses the construction $\langle f, g\rangle$.

There are multiple Q's, corresponding to different strategies of resolving these dependencies. To take a simple example from written English, suppose that the forms in X are strings. Sticking to single quotation marks, we can then define

(21) $$Qf\, x_{1\dots n} = \bar{\text{‘}} \frown \big(f(\bar{[} \frown x_1 \frown \bar{]}) \dots (\bar{[} \frown x_n \frown \bar{]})\big) \frown \bar{\text{’}}$$

where overlines cover literal strings and the operator $\frown$ denotes string concatenation. Given this Q, we can analyze the written form of (4) and (6) as follows.

(22) $(\lambda x.\, \overline{\text{The politician said she is}} \frown x)$
$(Q((\lambda x.\, \overline{\text{sorry to have used an}} \frown x)(Q\,\overline{\text{epithet}})))$
$= \overline{\text{The politician said she is 'sorry to have used an 'epithet''}}$

(23) $(\lambda x.\, \overline{\text{The politician admitted that she}} \frown x)$
$(Q(\lambda x_1 x_2.\, \overline{\text{lied}} \frown x_1 \frown \overline{\text{way into}} \frown x_2)\,\overline{\text{her}}\,\overline{\text{her job}})$
$= \overline{\text{The politician admitted that she 'lied [her] way into [her job]'}}$

The corresponding meaning components match the paraphrases given above under (4) and (6), *if* we assume the most obvious ways to resolve the anaphoric and presuppositional dependencies: the politician and her utterances (3) and (7), respectively. Some contexts make it more natural to resolve these dependencies in other ways, for example if the politician was asked by a talk show host whether 'you are sorry to have used an 'epithet'' or whether 'you lied your way into your job'. Whatever their context and however their dependencies are resolved, these examples demonstrate that we can analyze nested mixed quotes and mixed quotes of constructions.

My central claim is that the grammar of human language is largely generated by mixed-quote constructions. To be more precise, every primitive construction is a mixed quote (of the form (19)), a pure quote (such as $\langle Qf, f\rangle$), or a coinage (where all bets are off).

1.6. Mixed quotes of formal languages

For intuition, it may help to draw a parallel between this treatment of mixed quotation and the practice of code switching between natural and formal languages, such as embedding formulas in English sentences. In the examples below, we omit the quotation marks that some typographic conventions call for.

(24) $P \Rightarrow Q$ and P together entail Q.

(25) $\Gamma(2)$ contains 2.

(26) Alice said $\forall x \in \mathbb{R}.\, x^2 = -x^2$.

(27) Alice said what mathematicians use $\forall x \in \mathbb{R}.\, x^2 = -x^2$ to mean.

(28) Alice said $\Gamma(2)$ is negative.

(29) Alice said what mathematicians use $\Gamma(2)$ to mean is negative.

The embedding of formulas in the English sentences (24) and (25) are arguably cases of pure quotation: these sentences mention some formulas but do not use them. Our analysis of mixed quotation amounts to paraphrasing the mixed quotes in (26) and (28) in terms of the pure quotes in (27) and (29), respectively. These paraphrases preserve a de-re/de-dicto ambiguity: Alice's errors may be due to her ignorance about numbers or her ignorance about mathematical notation; the latter possibility requires interpreting the descriptions 'what mathematicians use $\forall x \in \mathbb{R}.\, x^2 = -x^2$ to mean' and 'what mathematicians use $\Gamma(2)$ to mean' de dicto.

Formulas can be quoted in a formal language as well as a natural language, for instance using *Gödel numbering*, a systematic mapping between formulas and integers. Given a Gödel numbering, the truth and provability of a formula in a logic can then be defined as arithmetic predicates. These predicates are the formal analogues of the meaning component of (19) and the paraphrases in (27) and (29). The ability to interpret one language in another enables linguistic creativity and reflection, so that first-order predicate logic may draw from not just the semantics but also the syntax of modal logic, so that Kolmogorov complexity is defined up to a constant, and so that the same low-level computer chip may run programs written in various high-level languages.

2. The prevalence of mixed quotation

In a mixed quote as in a pure quote, the quoted speaker may be generic, hypothetical, or institutional, and the quoted use may be generic, hypothetical, or habitual (Geurts and Maier 2003). Mixed quotation is thus a versatile source of constructions: in principle, the analysis in (19) gives rise to mixed quotes that draw their meanings from any construction use by any speaker, be it real or imagined, in the past, present, or future. It is thus perhaps unsurprising that mixed quotation can serve many purposes in the use and transmission of language.

2.1. Naming and other causes

A mild instance of prevalent mixed quotation is names according to a causal theory of reference (Kripke 1980). When Alice uses 'Aristotle' to mean Aristotle, unless she is baptizing Aristotle by coining the name, she relies on a previous use of the name to mean Aristotle. In other words, the nullary construction that pairs the name with the person is a mixed quote. This mixed quote is slightly unusual in two regards, but neither invalidates this analysis of names as mixed quotes.

i. The quoted form (say $Q\,\overline{\text{Aristotle}}$) and unquoted form (say $\overline{\text{Aristotle}}$) sound and look exactly the same, so one may be concerned as to how the hearer of Alice's utterance can know to parse 'Aristotle' as a quote. But there are preciously few ways for 'Aristotle' to appear in a English sentence, among which this parse is likely to be the top candidate.

ii. Alice and the other participants in the conversation may not recall a specific

occasion on which a specific speaker used the name to mean Aristotle, so the referent of x in (19) may be indeterminate. But like any other discourse referent, x can have its dependencies resolved as long as it is known that there exists a speaker (even an institutional one such as the English language) and a use (even a generic one such as usually). Such mixed quotes are common: the quote in (3) could be one, for example.

The use of 'Aristotle' that Alice mixed-quotes is either specifically the initial baptism of Aristotle or another mixed-quote of a use of 'Aristotle', and so on. The chain of naming occasions formed by mixed quotation is like a causal chain of naming, except a generic event does not usually appear on a causal chain. Research on reflection in programming languages (Smith 1982) shows how to compactly represent a long chain of mixed quotation

(30) !⌈!⌈!⌈... Aristotle ...⌉⌉⌉

with a stable meaning and a stable form. None of this discussion hinges on the actual existence of Aristotle; Sherlock Holmes would have done just as well.

If names are mixed quotes, they seem to take scope differently from ordinary mixed quotes (Michael Johnson, p.c.).

(31) Quine might have said that quotation 'has a certain anomalous feature'.

(32) It might have been the case that Aristotle was not named 'Aristotle'.

One reading of (31) does not entail that anyone used the words 'has a certain anomalous feature', so it seems possible to resolve the presupposition of the ordinary mixed-quote there within the scope of 'might'. In contrast, it does not seem that (32) has a false reading, so it seems necessary to resolve the presupposition that the name 'Aristotle' is used to mean someone outside the scope of 'might'.

When a linguist writes 'We assume the following notion of c-command ...' followed by the rest of a paper, that rest of the paper is a mixed quote of a speaker who uses 'c-command' to mean that notion, much as the sentence (2) could appear in a fairy tale that begins, 'Once upon a time, there was a president who likes to insert vowels when he pronounces words ...'.

Why stop at names and definitions? This 'copy-and-paste' syntax and semantics works across the board, so the sentence (33) can be cobbled together solely by composing mixed quotes as in (34). Ordinary language, then, is full of mixed quotes.

(33) Aristotle saw his sister.

(34) !⌈%[!⌈Aristotle⌉] saw %[!⌈%[!⌈him⌉]'s sister⌉]⌉

The punctuation in (34) notates walking up and down a tree of causation to curate forms and meanings from various speakers.

The analysis (34) assumes that the mixed-quoted expression 'him' is used to mean an anaphoric dependency. Similarly, in order for us to analyze Alice's use of 'I' to mean herself as a mixed quote of Bob's use of 'I' to mean himself, we must assume that the mixed-quoted use of 'I' means a context dependency on the first person, even though Bob also use the same form to mean himself.

If ordinary constructions such as '– saw –' are mixed quotes, they seem to allow wh-extraction and quantifying-in as in (35), which quotation is famed to not allow as in (36) (Quine 1953).

(35) a. Who did !⌈%[!⌈Aristotle⌉] see %[_]⌉?
b. !⌈%[!⌈Aristotle⌉] saw %[nobody]⌉

(36) a. Who said 'what' contains six letters?
b. $\exists x$. 'x' contains six letters.

Perhaps it is not impossible, merely difficult, for extraction and quantification to take place across a quote, especially a mixed quote. After all, examples such as (15) above are prima facie instances of quantifying into a quote, and the examples below suggest that wh-extraction is possible out of a mixed-quoted construction (Ernie Lepore, p.c.).

(37) What did Bush say Rove would !⌈protect us from ∼[_] in these turbulent times⌉?

(38) Who said Rove would !⌈protect us from ∼[what] in these turbulent times⌉?

2.2. Quantification and polarity

A quantifier can be thought of as a meta-construction, along the lines of abstract categorial grammars (de Groote 2002) and its noble line of intellectual ancestors: 'everybody' maps a unary construction to a nullary construction.

(39) $$\langle\ \lambda f.\, f\, \overline{\text{everybody}},\ \ \lambda g.\, \forall y.\, g\, y\ \rangle$$

This idea let us analyze 'everybody saw Mary' and 'Mary saw everybody', by applying (39) to the composition of '– saw –' and 'Mary'. However, (39) alone does not generate 'everybody saw everybody' because it only applies to unary constructions. To resolve this issue, it may seem natural to allow 'quantifying in' any argument position of an n-ary construction, mapping it to an $(n-1)$-ary construction (Hendriks 1993).

(40) $$\langle\ \lambda f\, x_{1\ldots k-1} x_{k+1\ldots n}.\, f\, x_{1\ldots k-1}\, \overline{\text{everybody}}\, x_{k+1\ldots n},\ \ \lambda g\, y_{1\ldots k-1} y_{k+1\ldots n}.\, \forall y.\, g\, y_{1\ldots k-1}\, y\, y_{k+1\ldots n}\ \rangle$$

However, an analogy between surface scope in quantification and left-to-right evaluation in other linguistic side effects such as anaphora and questions (Shan and Barker 2006) suggests only 'quantifying in' the last argument, that is, setting k above to n.

(41) $$\langle\ \lambda f\, x_{1\ldots n-1}.\, f\, x_{1\ldots n-1}\, \overline{\text{everybody}},\ \ \lambda g\, y_{1\ldots n-1}.\, \forall y.\, g\, y_{1\ldots n-1}\, y\ \rangle$$

If we analyze ‘everybody’ as (41) and ‘somebody’ analogously, then we generate only the surface-scope readings of (42) and (43).

(42) Somebody saw everybody.

(43) Everybody saw somebody.

Where does inverse scope come from, then? Mixed quotation offers one answer: we can generate inverse scope if we can quote the (wider) scope of the later quantifier as a construction, excluding that quantifier itself. For example, we can analyze (42) as (44) if we can mixed quote the unary construction ‘somebody saw –’, *hereby* used to mean the property of having been seen by somebody. The resulting interpretation can be glossed as (45) (which is coherent, unlike (46)—pace Quine (1960, 1953)).

(44) !⌈Somebody saw %[everybody]⌉

(45) For everybody y, the sentence ⌈Somebody saw %[y]⌉ is true.

(46) For everybody y, the sentence ⌈Somebody saw %[y]⌉ has eight letters.

To analyze a sentence with three quantifiers that take inverse scope with respect to each other, we would need nested mixed quotation. Perhaps some scope parallelism between quantifiers in discourse can be explained by the ease of quoting a construction recently used.

Because the more-quoted quantifier takes narrower scope in the example above, one might worry about cases where a mixed-quoted quantifier takes inverse scope over an unquoted quantifier, such as (47).

(47) The dean asked that a student ‘accompany every professor’. (Cumming 2003)

In fact, because written quotation marks may not indicate every level of actual quotation, we can treat such examples as long as we allow syntactic interjection, discussed in §1.3. Using the notation introduced there, the inverse-scope reading desired in (47) is analyzed in (48). Only the outermost pair of brackets in (48) manifests itself as a pair of quotation marks in (47).

(48) The dean asked that !⌈!⌈∼[%[⌈a student⌉]] accompany %[every professor]⌉⌉

A quick and dirty way to ‘evaluate’ quotation constructions such as (48) is to remove !⌈interpreted brackets enclosing no interjection⌉ and cancel out ∼[⌈syntactically interjected pure-quotes⌉] inside quotes.

This account of inverse scope lets us explain why polarity licensing requires not just that the licensor take scope over the licensee, but also that the licensor precede the licensee if they are clausemates (Ladusaw 1979). For example, whereas (49) has a surface-scope reading, (50) does not have an inverse-scope reading.

(49) Alice introduced nobody to anybody.

(50) *Alice introduced anybody to nobody.

Our explanation assumes that a clause like ‘Alice introduced anybody to Bob’ and a construction like ‘Alice introduced anybody to %[...]’ are not quotable, even though they can appear as part of a larger quotable item (for example when preceded by ‘nobody thinks’). Intuitively, they are not quotable because they are incomplete: they are unacceptable as utterances by themselves. This intuition can be enforced in one of two ways: either assign a different syntactic category or semantic type to a constituent that contains an unlicensed polarity item (Fry 1997), or always insert a licensor and a licensee in one fell meta-construction such as the following.

(51) $$\langle\ \lambda f.\, f\, \overline{\text{nobody}}\,\overline{\text{anybody}},\ \ \lambda g.\, \neg\exists yz.\, gyz\ \rangle$$

If there is no construction ‘Alice introduced anybody to %[...]’ to quote, then the strategy for generating inverse scope in (44) fails. This failure can be observed from the fact that the paraphrase (53) of (52), analogous to (45), is unacceptable.

(52) *!⌈Alice introduced anybody to %[nobody]⌉

(53) *For nobody y, the sentence ⌈Alice introduced anybody to %[y]⌉ is true.

References

Abbott, Barbara. 2003. Some notes on quotation. In *Hybrid quotations*, ed. Philippe de Brabanter, vol. 17(1) of *Belgian Journal of Linguistics*, 13–26. Amsterdam: John Benjamins.

Barker, Chris. 2007. Direct compositionality on demand. In *Direct compositionality*, ed. Chris Barker and Pauline Jacobson, 102–131. New York: Oxford University Press.

Cappelen, Herman, and Ernie Lepore. 2003. Varieties of quotation revisited. In *Hybrid quotations*, ed. Philippe de Brabanter, vol. 17(1) of *Belgian Journal of Linguistics*, 51–75. Amsterdam: John Benjamins.

Cumming, Samuel. 2003. Two accounts of indexicals in mixed quotation. In *Hybrid quotations*, ed. Philippe de Brabanter, vol. 17(1) of *Belgian Journal of Linguistics*, 77–88. Amsterdam: John Benjamins.

Davidson, Donald. 1979. Quotation. *Theory and Decision* 11(1):27–40.

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, CA: Morgan Kaufmann.

Geurts, Bart, and Emar Maier. 2003. Quotation in context. In *Hybrid quotations*, ed. Philippe de Brabanter, vol. 17(1) of *Belgian Journal of Linguistics*, 109–128. Amsterdam: John Benjamins.

de Groote, Philippe. 2002. Towards abstract categorial grammars. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 148–155. San Francisco, CA: Morgan Kaufmann.

Heim, Irene, and Angelika Kratzer. 1998. *Semantics in generative grammar*. Oxford: Blackwell.

Hendriks, Herman. 1993. Studied flexibility: Categories and types in syntax and semantics. Ph.D. thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Kripke, Saul A. 1980. *Naming and necessity*. Cambridge: Harvard University Press.

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.

Maier, Emar. 2007. Mixed quotation: Between use and mention. In *Proceedings of the 4th international workshop on logic and engineering of natural language semantics*, ed. Kei Yoshimoto. Japanese Society of Artificial Intelligence.

May, J. Peter. 1997. Definitions: Operads, algebras and modules. In *Operads: Proceedings of renaissance conferences (1995)*, ed. Jean-Louis Loday, James D. Stasheff, and Alexander A. Voronov, vol. 202 of *Contemporary Mathematics*, 1–7. Providence: American Mathematical Society.

Quine, Willard Van Orman. 1953. *From a logical point of view: 9 logico-philosophical essays*. Cambridge: Harvard University Press.

———. 1960. *Word and object*. Cambridge: MIT Press.

Shan, Chung-chieh, and Chris Barker. 2006. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy* 29(1):91–134.

Smith, Brian Cantwell. 1982. Reflection and semantics in a procedural language. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Also as Tech. Rep. MIT/LCS/TR-272.

Steedman, Mark J. 1996. *Surface structure and interpretation*. Cambridge: MIT Press.

Taha, Walid. 2004. A gentle introduction to multi-stage programming. In *International seminar on domain-specific program generation (2003)*, ed. Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, 30–50. Lecture Notes in Computer Science 3016, Berlin: Springer.

Exploits of a mom http://xkcd.com/327/ (2007)