Representing Covert Movements by Delimited Continuations

Daisuke Bekki and Kenichi Asai

Department of Information Science, Faculty of Science,
Ochanomizu University,
Ootsuka 2-1-1, Bunkyo-ku, Tokyo, Japan

1 Background

1.1 Motivation: Covert Movements and Delimited Continuations

In phenomena which have been claimed to require "covert movements" in generative terms, a relevant lexical item seems to require a means to somehow refer to the meaning of its *surroundings* in order for the meaning of the whole sentence to be properly computed. This has motivated generative/transformational grammars to adopt a movement of the relevant item to the position where its scope contains surroundings that influence its meaning, while it remains as an issue to be solved for categorial/Lambek-style grammars, namely, grammars without movements.

Analyses that represent "covert movements" by means of *delimited continuations* (Danvy and Filinski (1990)) have recently attracted much attention in the field of natural language semantics (de Groote (2001), Shan (2002), Barker (2002), Barker (2004), Shan and Barker (2006), Barker and Shan (2006), Shan (2007), Otake (2008)). Delimited continuation is a notion which originates in the theory of functional programming languages; it enables each subterm to refer to "the rest of the computation", i.e. its surroundings.

In this paper, we propose a new method to define delimited continuations in terms of an *internal monad*, which is described by *Meta-Lambda Calculus* (cf. Bekki (2009)), which thereby enables us to analyse linguistic phenomena such as focus and inverse scope. In the following sections, we will review the notion of continuations both in programming and natural languages, then point out some problems that the preceding continuation theory of natural language semantics, especially in what is called *Continuized Semantics*, advocated in Barker (2002) and Barker (2004), and demonstrate how our approach successfully overcomes those problems. In the appendix, we will briefly present the language and theory of Meta-Lambda Calculus used throughout the paper.

1.2 Continuations and CPS Transformation

The concept of a Continuation (Stratchey and Wadsworth (1974)) was devised originally to capture the semantics of control operators such as jumps, which otherwise do not fit into a purely functional view of programming languages.

K. Nakakoji, Y. Murakami, and E. McCready (Eds.): JSAI-isAI, LNAI 6284, pp. 161–180, 2010.

For any subterm $N : \alpha$ of a lambda term $M : O$, a continuation for N is a function $k : \alpha \rightarrow O$, which returns $M : O$ when $N : \alpha$ is passed as its argument. Intuitively, this function corresponds to the *rest* of the computation for N in M.

It is often the case that the whole term M is not specified and only its type, which is called the **answer type**, is implicitly given. Let us use the following notation, using T as a type operator:

$$T\alpha \stackrel{def}{\equiv} (\alpha \rightarrow O) \rightarrow O$$

where O is an answer type implicitly assumed.

Example 1. In the term $1+(a*2) : int$ (the answer type is int), the continuation of a subterm $a : int$ is a function $\lambda\kappa.(1 + (\kappa * 2)) : (int \rightarrow int) \rightarrow int$.

Example 2. In the proposition $love(john, mary) : t$ (the answer type is t), the continuation of a subterm $mary : e$ is a function $\lambda\kappa.love(john, \kappa) : (e \rightarrow t) \rightarrow t$.

In order for each subterm to refer to its continuation, terms are transformed using the continuation-passing style (CPS).

Definition 3 (The transformation rules for call-by-value CPS). *(Slightly adapted from the definition originally proposed in Plotkin (1975).)*

$$\begin{aligned}
[\![x]\!] &\stackrel{def}{\equiv} \lambda\kappa.\kappa(\overline{x}) \\
[\![fM]\!] &\stackrel{def}{\equiv} \lambda\kappa.[\![M]\!](\lambda a.\kappa(\overline{f}a)) \\
[\![\lambda x.M]\!] &\stackrel{def}{\equiv} \lambda\kappa.\kappa(\lambda\overline{x}.[\![M]\!]) \\
[\![MN]\!] &\stackrel{def}{\equiv} \lambda\kappa.[\![M]\!](\lambda f.[\![N]\!](\lambda a.(fa)\kappa))
\end{aligned}$$

$\overline{x}$ is just the variable x but assigned a different type: if the type of x is α, then the type of $\overline{x}$ is α^*, where the operator * is recursively defined as follows:

Definition 4 (* operator).

$$\begin{aligned}
b^* &\mapsto b \\
(\alpha \rightarrow \beta)^* &\mapsto (\alpha^* \rightarrow T(\beta^*))
\end{aligned}$$

By CPS transformation, a lambda term of type α is uniformly transformed into one of type $T(\alpha^*)$. It should be noted that a linguist who is familiar with the continuation analysis of Barker (2002) and Barker (2004) must note that the CPS of value types and functional types are of different type.

Example 5.

$$\begin{aligned}
e^* &= e \\
(e \rightarrow t)^* &= (e \rightarrow Tt) \\
(e \rightarrow e \rightarrow t)^* &= (e \rightarrow T(e \rightarrow Tt))
\end{aligned}$$

Not all terms in CPS have their counterparts in terms of the direct style. Such terms include a term in which there is reference to its continuation. This is the power resulting from the CPS transformation and an example of such an extended term is the `shift/reset` operator (Danvy and Filinski (1990)), which is a static way of capturing these operators.

Definition 6 (Shift/Reset operators)

$$[\![\mathit{shift}\ f.M]\!] \stackrel{def}{\equiv} \lambda\kappa.([\![M]\!](\lambda x.x)[\lambda a.\lambda\kappa'.\kappa'(\kappa a)/\overline{f}])$$
$$[\![\mathit{reset}(M)]\!] \stackrel{def}{\equiv} \lambda\kappa.\kappa([\![M]\!](\lambda x.x))$$

1.3 Barker's Continuized Semantics for Natural Language

Barker (2001) and de Groote (2001) first pointed out the similarity between Montague-style type raising and CPS transformation. Barker (2002) and Barker (2004) regarded the type raising operation as an instance of translation into CPS, and this is the strategy that succeeding analyses (Shan and Barker (2006), Barker and Shan (2006) have adopted. Let us call this approach *Continuized Semantics* after Barker (2002). Barker's CPS transformation rules are defined as follows.

Definition 7 (The transformation rules for call-by-value CPS). *(Slightly adapted from the definition proposed in Barker (2002) and Barker (2004).)*

$$[\![x]\!] \stackrel{def}{\equiv} \lambda\kappa.\kappa(x)$$
$$[\![f]\!] \stackrel{def}{\equiv} \lambda\kappa.\kappa(f)$$
$$[\![MN]\!] \stackrel{def}{\equiv} \lambda\kappa.[\![M]\!](\lambda f.[\![N]\!](\lambda a.\kappa(fa))) \ \textit{(Call-by-value)}$$
$$[\![MN]\!] \stackrel{def}{\equiv} \lambda\kappa.[\![M]\!](\lambda f.[\![N]\!](\lambda a.\kappa(fa))) \ \textit{(Call-by-name)}$$

While Plotkin's original definition in Definition 3 transforms a lambda term of type α to a lambda term of type $T(\alpha^*)$, Barker's rules transform a lambda term of type α to a lambda term of type $T(\alpha)$. Taking the complexity of the star operator (Definition 4) into account, Barker's transformation rules are substantially simpler, and contain substantial flaws as well. We will come back to this point in Section 2.2.

In Barker (2004), quantifiers such as *everyone* and *someone* are assigned their semantic representations in CPS style in the lexicon as shown below.

$$\text{everyone} \vdash \lambda\kappa.\forall x.\kappa x : (e \to t) \to t$$
$$\text{someone} \vdash \lambda\kappa.\exists x.\kappa x : (e \to t) \to t$$

With the presence of the type raising operation (indicated by (T)), the type mismatch problem can be resolved as in the following derivation.

(1)

$$\dfrac{\dfrac{\text{somebody}}{\lambda P.\exists x(Px)} \quad \dfrac{\dfrac{\text{loves}}{\lambda y.\lambda x.love(x,y)} \quad \dfrac{\dfrac{\text{everybody}}{\lambda P.\forall y(Py)}}{\lambda P.\lambda x.\forall y(Pyx)}\,(T)}{\lambda x.\forall y(love(x,y))}\,(<)}{\exists x(\forall y(love(x,y)))}\,(>)$$

On the other hand, Continuized Semantics achieves the same effect in the following way:

(2)

$$\dfrac{\dfrac{\text{somebody}}{\lambda\kappa.\forall x(\kappa x)} \quad \dfrac{\dfrac{\text{loves}}{\lambda\kappa.\kappa(\lambda y.\lambda x.love(x,y))} \quad \dfrac{\text{everybody}}{\lambda\kappa.\exists y(\kappa y)}}{\lambda\kappa.\forall y(\kappa(\lambda x.love(x,y)))}\,(>CBN)}{\lambda\kappa.\exists x(\forall y(\kappa(love(x,y))))}\,(<CBN)$$

Barker claims that this method uniformly yields an NP, be it a proper name or quantifier, a representation of type e without recourse to type raising. Moreover, a boolean type approach to coordination (Partee and Rooth (1983), Winter (2001), among many others) can be integrated into this system as an instance of CPS.

2 Problems of Continuized Semantics

In this section, we will show that Continuized Semantics implies at least four problems that need to be addressed; the first three are theoretical, and the last one is empirical.

2.1 `fcontrol` and `run`

Barker (2004) mentioned the interaction between the adverbial *only* and focus within its scope as an instance of applying delimited continuation to natural language semantics, which can be defined by using `run` and `fcontrol` operators (Sitaram and Felleisen (1990), Sitaram (1993)) as follows (slightly adapted).

$$[\![only\ P]\!] \stackrel{def}{\equiv} \texttt{run}(P)(\lambda x.\lambda\kappa.\lambda y.(\kappa xy \wedge \forall z.(\kappa zy \rightarrow x = z)))$$
$$[\![[M]_F]\!] \stackrel{def}{\equiv} \texttt{fcontrol}([\![M]\!])$$

It is clear that the behaviour of Sitaram's `fcontrol` and `run` fits in with this phenomenon. However, this analysis is defective in the sense that Barker has not given any definition of those operators in terms of his CPS.

CPS transformation is regarded as a static simulation of control operators and Sitaram's `fcontrol`/`run` operators (or, equivalently, Felleisen (1988)'s `control`/`prompt` operators) are defined only dynamically, and it is still controversial whether they can be statically represented in terms of CPS.

Recently, Dybvig et al. (2007) has given a *small step semantics* for `control`/`prompt` operators, and mentions its relation to CPS (in section 2.3.4); nevertheless that relation is still far from obvious at present. Fairly speaking, Continuized Semantics has left much work to be done, including the definition of `fcontrol`/`run` operators, in order to argue that it has good prospects for the analysis of adverbial *only* and focus.

2.2 Lambda Abstractions

As we mentioned in Section 1.3, the simplification of the transformation rules in Continuized Semantics comes at a cost; there is no rule for transforming a lambda abstraction term. According to Barker's rule, a lambda term $\lambda x.M$ of the type $\alpha \to \beta$ is transformed into one of type $T(\alpha \to \beta)$, namely, $((\alpha \to \beta) \to O) \to O$. Then it seems natural to assume that this term has the form $\lambda\kappa.N$, where κ is a variable of type $(\alpha \to \beta) \to O$ and N is some lambda term of type O, and it also seems natural to assume that N is constructed only from a combination of the following components (where $[\![M]\!]$ contains x as a free variable).

$$\kappa : (\alpha \to \beta) \to O$$
$$x : \alpha$$
$$[\![M]\!] : (\beta \to O) \to O$$

But this does not seem to be achievable[1]. The main difficulty lies in the impossibility of obtaining a term of type β from $[\![M]\!]$.[2]

Barker defends the fact that his theory lacks a transformation rule for a lambda abstraction term by claiming that it might be possible to eliminate any lambda abstraction terms from logical forms of natural language. For example, Barker's mechanism can deal with the type mismatch problem of quantifiers without recourse to quantifier raising (QR), and Shan and Barker (2006) mentions a treatment of relative clauses without operator movements.

However, in natural language semantics, lambda abstractions are more abundant than his optimism allows for. We may even say that formal semantics

[1] For example, the term $\lambda\kappa.[\![M]\!](\lambda v : \beta.\kappa(\lambda x : \alpha.v))$ is of type $T(\alpha \to \beta)$. But this term contains x as a free variable and the inner variable x is vacuous, which obviously does not serve as a translation of $\lambda x.M$. This argument is related to our realization of the transformation rule of a lambda abstraction term via meta-lambda calculus. See Section 4.2 for details.

[2] This is only possible in the special case in which the answer type is t; then $\bigwedge[\![M]\!]$ is of type β if β is a complete lattice.

without lambda abstraction is bound hand and foot: How can we formalize comparatives without lambda abstractions? How can we extend our system with event variables and possible worlds without lambda abstractions?

2.3 The Type of Determiners

Another theoretical problem of Continuized Semantics concerns the type that it assigns to determiners. According to the standard theory of generalized quantifiers (in direct style), a determiner (e.g. "every") has a type $(e \to t) \to (e \to t) \to t$, and a common noun (e.g. "man") has a type $e \to t$.

$$\text{every} \vdash \lambda Q.\lambda P.\forall x(Px \to Qx) : (e \to t) \to (e \to t) \to t$$
$$\text{man} \vdash man : e \to t$$

But this view, as it stands, does not fit into Continuized Semantics for the following reasons:

1. In the first place, the semantic representation of "every" is of lambda abstraction form, therefore it cannot be transformed into CPS.
2. Suppose that the representation of "every" is somehow transformed into CPS; it should have type $T((e \to t) \to (e \to t) \to t)$. Suppose further that the representation of "man" is transformed with CPS into one whose type is $T(e \to t)$. Then the composition of these two yields the representation of "every man" whose type is supposed to be $T((e \to t) \to t)$. However, this is not consistent with one of the selling points of Continuized Semantics; namely, that the representation of "everyone" is of type $T(e) \equiv (e \to t) \to t$ and already in CPS form. This means that "every man" and "every" have different types!

In order to get around this problem, Barker (2002) assumes that the representation of a determiner such as "every" is of type $T((e \to t) \to e)$, namely, $(((e \to t) \to e) \to t) \to t$, instead of $T((e \to t) \to (e \to t) \to t)$.

The first question that naturally comes to mind is whether one can define a representation of generalized quantifiers having this type, since this intuitively means that we define the representation of "every" as having type $(e \to t) \to e$: in other words, a function that takes a set like "man" and returns an object of type e.

This seems impossible within the compass of direct semantics. But Continuized Semantics gets round this problem by employing choice functions of type $(e \to t) \to e$ (cf. Kratzer (1998)). Determiners are defined in the appendix of Barker (2002) as follows, in which the variable f is a choice function that plays the role of choosing one entity from a given set, and what is universally quantified here is this f.

$$\text{every} \vdash \lambda d.\forall f(d(f)) : T((e \to t) \to e)$$
$$\text{some} \vdash \lambda d.\exists f(d(f)) : T((e \to t) \to e)$$

The derivation goes as follows. The resulting representation states that for any of those choice functions, $\kappa(f(man))$ is true.

(3)

$$(>CBV)\dfrac{\dfrac{\text{every}}{\lambda d.\forall f(d(f)) : T((e \to t) \to e)} \quad \dfrac{\text{man}}{\lambda\kappa.\kappa(man) : T(e \to t)}}{\lambda\kappa.\forall f(\kappa(f(man))) : T(e)}$$

Generalized quantifiers like "most" are also definable in terms of a choice function.[3]

$$\text{most} \vdash \lambda d.\exists C \in \mathbf{MOST}(\forall f \in C\ (d(f))) : T((e \to t) \to e)$$
$$\mathbf{MOST} \stackrel{def}{\equiv} \{C \mid \forall P(|P| < 2 * |\{x \mid \exists f \in C\ (x = f(P))\}|)\}$$

However, there are determiners, such as *only* and *even*, which cannot be given a representation of this type. Let us take "only" for example. Putting aside the pre-suppositional content of "only", the following representation is plausible for "only" in direct style.

$$\text{only} \vdash \lambda z.\lambda P.(Pz \wedge \forall x(Px \to x = z)) : e \to (e \to t) \to t$$

This means that "only" must be represented as a type $T(e \to e)$ in Continuized Semantics, because the derivation must go as follows.

(4)

$$(>CBV)\dfrac{\dfrac{\text{only}}{T(e \to e) \equiv ((e \to e) \to t) \to t} \quad \dfrac{\text{John}}{T(e) \equiv (e \to t) \to t}}{T(e) \equiv (e \to t) \to t}$$

Now it seems impossible to define a representation of "only" as having type $T(e \to e)$, considering its propositional content.[4] The same argument applies to "even". Moreover, there are other kinds of determiners which lead to the same sort of difficulty, such as "same" or "different", among many others.

Therefore, the central view of Continuized Semantics that type raising can be reduced to CPS transformations, although appealing at a glance, cannot be maintained as it stands if we consider a wider range of phenomena in natural language semantics.

[3] Barker (2002) and Barker (2004) did not mention anything about downward entailing quantifiers, but they are also definable in terms of negated forms of upward entailing quantifiers as follows:

$$\text{few} \vdash \lambda d.\neg\exists C \in \mathbf{MOST}(\forall f \in C\ (d(f))) : T((e \to t) \to e)$$

[4] Barker (2004) includes a lexical entry for adverbial "only" which is defined via the `run` operator. But this does not remedy the above problem; in the sentence "only John runs", "John" is focused and marked with the $[\ \ ]_F$ operator, but since the continuation that "John" receives is empty, "John" cannot refer to the representation of the verb phrase.

2.4 Inverse Scope

Barker (2002) suggests that the scope ambiguity between linear and inverse scope readings (in sentences with more than two quantifiers) can be reduced to a choice between two evaluation strategies: *call-by-value* and *call-by-name* in the interpretation of functional applications.

For example, the derivation of the sentence "Everybody loves somebody" in (2) uses only the call-by-name strategy for the interpretation of functional applications, which yields the linear scope reading. On the other hand, by using only the call-by-value strategy, the inverse scope reading yields the interpretation shown by the following derivation.

(5)

$$\dfrac{\dfrac{\text{somebody}}{\begin{array}{c}\lambda\kappa.\exists x(\kappa x)\\ :T(e)\end{array}}\ (<CBV)\ \dfrac{\dfrac{\text{loves}}{\begin{array}{c}\lambda\kappa.\kappa(\lambda y.\lambda x.love(x,y))\\ :T(e\rightarrow e\rightarrow t)\end{array}}\ \dfrac{\text{everybody}}{\begin{array}{c}\lambda\kappa.\forall y(\kappa y)\\ :T(e)\end{array}}}{\begin{array}{c}\lambda\kappa.\forall y(\kappa(\lambda x.love(x,y)))\\ :T(e\rightarrow t)\end{array}}\ (>CBV)}{\begin{array}{c}\lambda\kappa.\forall y(\exists x(\kappa(love(x,y))))\\ :T(t)\end{array}}$$

It is true that its conception and repercussions (we mean, if natural language has any phenomena which can be explained by assuming that there are two different strategies, and more importantly, we have the chance to *choose* one of them, for each evaluation of a functional application) are worth pursuing, as Shan and Barker (2006) does.

However, we believe that any analysis of scope ambiguity has to be verified by sentences which contain more than three quantifiers. This is not a flaw only of Continuation Semantics, but rather is an inherited vice in the field of formal semantics, which has unquestioningly assumed that there exists a reading for every combination of quantifiers contained in a given sentence; namely, $n!$ different readings for a sentence with n quantifiers, based only on the fact that a sentence which contains only two quantifiers has two different readings: linear and inverse scope readings. But the situation becomes different when $n > 3$.

For example, more deliberate studies of scope ambiguity, such as Hayashishita (2003) among others, point out that a sentence with three quantifiers, Q_1, Q_2 and Q_3 in their linear order, hardly shows the $Q_3 > Q_2 > Q_1$ reading (which we may call a "reversed reading"), such as the *every* > *most* > *some* reading in the following sentence.

(6) Some teachers introduce most students to every company.

The analysis of Barker (2002) and Barker (2004) wrongly generates the reversed reading for the above sentence, if every functional application is evaluated by the call-by-value strategy.

On the other hand, the *intermediate*-inverse scope readings such as *most* > *some* > *every* or *every* > *some* > *most* seem to exist, but these are not derivable

in the analysis of Barker (2002) nor Barker (2004), because there are only two choices in the scope relation betwen *most* and *every*, namely *most* > *every* or *every* > *most*, and the scope of the subject is either higher than both of them, or lower than both of them, yielding only the following four combinations (the fourth of which is the reversed reading).

$$\begin{aligned} &\textit{some} > \textit{most} > \textit{every} \\ &\textit{most} > \textit{every} > \textit{some} \\ &\textit{some} > \textit{every} > \textit{most} \\ &\textit{every} > \textit{most} > \textit{some} \end{aligned}$$

Thus the analysis of Continuized Semantics does not properly predict the empirical combination of linear/intermediate-inverse/inverse scope readings. This will be solved in our analysis, in which scope ambiguity is not due to the choice of evaluation strategy, but rather to the use of the *inverse scope operator*; See Section 4.4.

2.5 Summary of the Problems

To summarize, Continuized Semantics has the following set of problems:

- Lack of definitions for `fcontrol` and `run` in terms of CPS
- Empirically wrong predictions for inverse scope readings
- No transformation rule for lambda abstraction construction
- Existence of undefinable determiners

We believe that these problems are particular to the formulation of Continuized Semantics and not problems for the use of continuations in general for natural language semantics.

The central idea of Continuized Semantics is to regard CPS transformation as a generalization of type raising operations and thus it enables us to do without type raising operations. In other words, Continuized Semantics represents the whole semantic system in CPS, in order to obviate the need for type raising.

This view is the root cause of the problem that we discussed in Section 2.3, but we think that it is burdened by a more conceptual problem; can we really consider Continuized Semantics as doing without type raising operations? As the conclusion of Barker (2002) stated, CPS transformation can also be seen as a kind of raising. If this is so, then it can be regarded as raising everything by CPS transformation!

Moreover, every semantic representation in Continuized Semantics is described in CPS. This opposes the original concept of CPS transformation in Danvy and Filinski (1990) , in which the advantage of CPS transformation is that we can write representations in *direct style* while making use of control operators such as `shift`/`reset`, whose interpretations are defined in terms of CPS; nevertheless they are allowed to appear within other representations in direct style.

The achievement of Barker (2002) and Barker (2004) was to introduce the notion of (delimited) continuations to the field of natural language semantics and indicate that continuations may have interactions with many linguistic phenomena in an important way. At the same time, its motivation to reduce type raising operations has been misunderstood to be the central merit of continuations, which leads to a situation where the real descriptive power of continuations in linguistics is hard to grasp.

Unlike Continuized Semantics, our strategy is to write semantic representations in direct style according to the line of Danvy and Filinski (1990) . Then we will define control operators in terms of CPS transformations, allowing them to appear among direct style representations. This strategy places an emphasis on bringing out the original expressive power of delimited continuations.

In the next section, we will present our proposal in the following way.

1. We define an *internal monad* for delimited continuations in term of the *meta-lambda calculus* (Bekki (2009)). This enables delimited continuations to co-exist with other monadic analyses in Bekki (2009).
2. We define CPS transformation (of simply-typed lambda terms) by means of internal monads. This transformation, unlike Barker's, is defined generally enough to treat any lambda terms, including lambda abstraction constructions.
3. We will demonstrate how the resulting analysis solves the problems we have pointed out in this section: the problem of "only" and focus, the transformation of lambda abstraction terms, the problem of types of determiners, and the empirical problem of inverse scope readings.

3 Proposal: Delimited Continuations via Meta-Lambda Calculus

3.1 Transformation Rules by Continuation Monad

To begin, we define a general translation rule in terms of *internal monads* (Bekki (2009)p.202) which is robust enough to deal with any term defined in typed lambda calculus.

Definition 8 (Internal monad for deliminted continuations). *The internal monad for delimited continuations is a triple* $\langle \boldsymbol{T}, \eta, \mu \rangle$*, each of which is defined using the following meta-lambda terms.*

$$\begin{aligned} \boldsymbol{T} &= \zeta f.\zeta X.\zeta \kappa.(X \lhd (\zeta v.\kappa \lhd (f \lhd v))) \\ \eta &= \zeta X.\zeta \kappa.(\kappa \lhd X) \\ \mu &= \zeta X.\zeta \kappa.(X \lhd (\zeta v.v \lhd \kappa)) \end{aligned}$$

Definition 9 gives a set of transformation rules for simply-typed lambda terms, which are parametrized by internal monads.

Definition 9 (Transformation with Internal Monad (call-by-value))

$$\begin{aligned}
[\![x]\!]_{\boldsymbol{T}} &= \eta \triangleleft x \\
[\![c]\!]_{\boldsymbol{T}} &= \eta \triangleleft c \\
[\![\lambda x.M]\!]_{\boldsymbol{T}} &= (\boldsymbol{T} \triangleleft (\zeta X.\lambda x.X)) \triangleleft [\![M]\!]_{\boldsymbol{T}} \\
[\![MN]\!]_{\boldsymbol{T}} &= \mu \triangleleft (\boldsymbol{T} \triangleleft (\zeta X.((\boldsymbol{T} \triangleleft (\zeta Y.XY)) \triangleleft [\![N]\!]_{\boldsymbol{T}}) \triangleleft [\![M]\!]_{\boldsymbol{T}})
\end{aligned}$$

Definition 10 (Translation for continuation monad). *Given an internal monad for continuations, the translation rules for lambda terms with delimited continuations are accordingly defined as follows.*

$$\begin{aligned}
[\![x]\!]_c &= \zeta\kappa.(\kappa \triangleleft x) \\
[\![c]\!]_c &= \zeta\kappa.(\kappa \triangleleft c) \\
[\![\lambda x.M]\!]_c &= \zeta\kappa.([\![M]\!]_c \triangleleft (\zeta v.\kappa \triangleleft (\lambda x.v))) \\
[\![MN]\!]_c &= \zeta\kappa.([\![M]\!]_c \triangleleft (\zeta m.[\![N]\!]_c \triangleleft (\zeta n.\kappa \triangleleft (mn)))
\end{aligned}$$

3.2 Control Operators

In the light of this setting, shift/reset operators are defined in the following way.

Definition 11 (Control operators).

$$\begin{aligned}
[\![\mathit{shift}\ \kappa.M]\!] &= \zeta\kappa.([\![M]\!] \triangleleft (\zeta x.x)) \\
[\![\mathit{reset}(M)]\!] &= \zeta\kappa.(\kappa \triangleleft ([\![M]\!] \triangleleft (\zeta x.x)))
\end{aligned}$$

The definition above looks too simple compared with that of Danvy and Filinski (1990) but has the same effects, as indicated by the following computations.

$$\begin{aligned}
[\![1 + \mathit{reset}(10 + \mathit{shift} f.(f(f(100))))]\!] &= \zeta\kappa.(\kappa \triangleleft 121) \\
[\![1 + \mathit{reset}(10 + \mathit{shift} f.(100))]\!] &= \zeta\kappa.(\kappa \triangleleft 101) \\
[\![1 + \mathit{reset}(10 + \mathit{shift} f.(f(100) + f(1000)))]\!] &= \zeta\kappa.(\kappa \triangleleft 1121)
\end{aligned}$$

4 Solutions

4.1 "Focus Movement" as `shift/reset`

Rooth (1992) discussed the truth conditions of the two sentences (7a) and (7b) which differ only in the location of the focus. As a description of the situation where Mary introduced Bill and Tom to Sue, with no other introductions, (7a) is false and (7b) is true.

(7) a. Mary only introduced $[\text{Bill}]_F$ to Sue.
b. Mary only introduced Bill to $[\text{Sue}]_F$.

In order to account for this, Wagner (2006), among others, adopts an operation called "focus movement", which is an instance of covert movements.

We can, however, directly compute the semantic representation without covert movements under the mechanism of delimited continuations with `shift/reset` operators. This is achieved by the following definition, where focus is interpreted by means of the `shift` operator, and the adverbial "only" is the `reset` operator.

Definition 12 (Focus operator). *For any meta-lambda term* $\phi : e$ *and* $\psi : (e \to t) \to e \to t$,

$$[\phi]_F \stackrel{def}{\equiv} \mathit{shift}\ f.\lambda x.\forall z((f \triangleleft z) \triangleleft x \to z = \phi) : (e \to e \to t) \to e \to t$$
$$\mathit{only}(\phi) \stackrel{def}{\equiv} \mathit{reset}(\phi) : (e \to t) \to e \to t$$

The semantic representations for (7a) and (7b) are respectively calculated as follows:

$$\begin{aligned}
&[\![((\mathit{only}\ ((\mathit{introduce}\ [b]_F)\ s)\ m)]\!]_c \\
&= \zeta\kappa.[\![\mathit{only}\ ((\mathit{introduce}\ [b]_F)\ s)]\!]_c(\zeta x.\kappa(x(m))) \\
&= \zeta\kappa.\kappa([\![(\mathit{introduce}\ [b]_F)\ s]\!]_c(\zeta x.x)(m)) \\
&= \zeta\kappa.\kappa([\![[b]_F]\!]_c(\zeta w.\mathit{introduce}(w)(s)))(m) \\
&= \zeta\kappa.\kappa([\![b]\!]_c(\lambda y.\forall z(\mathit{introduce}(z)(s)(m) \to z = y)) \\
&= \zeta\kappa.\kappa(\forall z(\mathit{introduce}(z)(s)(m) \to z = b))
\end{aligned}$$

$$\begin{aligned}
&[\![((\mathit{only}\ ((\mathit{introduce}\ b)\ [s]_F)\ m)]\!]_c \\
&= \zeta\kappa.[\![\mathit{only}\ ((\mathit{introduce}\ b)\ [s]_F)]\!]_c(\zeta x.\kappa(x(m))) \\
&= \zeta\kappa.\kappa([\![(\mathit{introduce}\ b)\ [s]_F]\!]_c(\zeta x.x)(m)) \\
&= \zeta\kappa.\kappa([\![[s]_F]\!]_c(\zeta w.\mathit{introduce}(b)(w)))(m) \\
&= \zeta\kappa.\kappa([\![s]\!]_c(\zeta y.\forall z(\mathit{introduce}(b)(z)(m) \to z = y)) \\
&= \zeta\kappa.\kappa(\forall z(\mathit{introduce}(b)(z)(m) \to z = s)))
\end{aligned}$$

4.2 Transforming Lambda Abstractions

The transformation rules in Definition 9 properly transform a lambda term of lambda abstraction form in spite of the difficulty we mentioned in Section 2.2. The rules in Definition 9 transform a lambda term of the form $\lambda x.[\![M]\!]$ into the following form:

$$\zeta\kappa.([\![M]\!]_c \triangleleft (\zeta v.\kappa \triangleleft (\lambda x.v)))$$

Notice this form resembles the one we mentioned in the footnote in Section 2.2, which is as follows.

$$\lambda\kappa.[\![M]\!](\lambda v.\kappa(\lambda x.v))$$

Recall that this form does not serve our purpose, because 1) the variable x remains free in $[\![M]\!]$, and 2) the inner variable x is vacuous. However, in our settings, the free variable x in $[\![M]\!]$ gets bound by the inner lambda operator. This

is due to the difference of operators binding the variable v; in the failed transformation, v is bound by a *normal* lambda operator, while in our transformation, the variable v is bound by a *meta-lambda* operator, thus v is actually a *meta-variable*, which keeps track of the free variables that it contains. This enables the inner lambda operator to bind the free variable contained in the meta-variable v. For a detailed definition of the objects in meta-lambda calculus, see Appendix A. Here we show only the typing of the transformation of lambda abstraction terms below.

Example 13

$$\dfrac{\dfrac{\dfrac{}{\kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O), v : (\Gamma, x : \alpha \vdash \beta) \Vdash \kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O)}(MVAR) \quad \dfrac{\dfrac{}{\kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O), v : (\Gamma, x : \alpha \vdash \beta) \Vdash v : (\Gamma, x : \alpha \vdash \beta)}(MVAR)}{\kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O), v : (\Gamma, x : \alpha \vdash \beta) \Vdash \lambda x.v : (\Gamma \vdash \alpha \to \beta)}(LAM)}{\kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O), v : (\Gamma, x : \alpha \vdash \beta) \Vdash \kappa \triangleleft (\lambda x.v) : (\Gamma \vdash O)}(MAPP)}{\kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O) \Vdash \zeta v.\kappa \triangleleft (\lambda x.v) : (\Gamma, x : \alpha \vdash \beta) \Rightarrow (\Gamma \vdash O)}(MLAM)$$

$$\dfrac{\dfrac{\kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O) \Vdash [\![M]\!]_c : ((\Gamma, x : \alpha \vdash \beta) \Rightarrow (\Gamma \vdash O)) \Rightarrow (\Gamma \vdash O) \quad \kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O) \Vdash \zeta v.\kappa \triangleleft (\lambda x.v) : (\Gamma, x : \alpha \vdash \beta) \Rightarrow (\Gamma \vdash O)}{\kappa : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O) \Vdash [\![M]\!]_c \triangleleft (\zeta v.\kappa \triangleleft (\lambda x.v)) : (\Gamma \vdash O)}(MAPP)}{\Vdash \zeta\kappa.[\![M]\!]_c \triangleleft (\zeta v.\kappa \triangleleft (\lambda x.v)) : (\Gamma \vdash \alpha \to \beta \Rightarrow \Gamma \vdash O) \Rightarrow (\Gamma \vdash O)}(MLAM)$$

4.3 Transforming Determiners

Determiners such as "every" and "only", which have proved to be problematic for Continuized Semantics, are successfully transformed in the following way.

$$\begin{aligned}
[\![every]\!] &= [\![\lambda P.\lambda Q.\forall x(Px \to Qx)]\!] \\
&= \zeta\kappa.[\![\lambda Q.\forall x(Px \to Qx)]\!](\zeta v.\kappa(\lambda P.v)) \\
&= \zeta\kappa.(\zeta\kappa'.[\![\forall x(Px \to Qx)]\!](\zeta v'.\kappa'(\lambda Q.v')))(\zeta v.\kappa(\lambda P.v)) \\
&= \zeta\kappa.(\zeta\kappa'.(\zeta\kappa''.[\![Px \to Qx]\!](\zeta v''.\kappa''(\forall x.v')))(\zeta v'.\kappa'(\lambda Q.v')))(\zeta v.\kappa(\lambda P.v)) \\
&= \zeta\kappa.(\zeta\kappa'.(\zeta\kappa''.\zeta\kappa'''.\kappa'''(Px \to Qx)(\zeta v''.\kappa''(\forall x.v')))(\zeta v'.\kappa'(\lambda Q.v')))(\zeta v.\kappa(\lambda P.v)) \\
&= \zeta\kappa.(\zeta\kappa''.\zeta\kappa'''.\kappa'''(Px \to Qx)(\zeta v''.\kappa''(\forall x.v')))(\zeta v'.(\zeta v.\kappa(\lambda P.v))(\lambda Q.v')) \\
&= \zeta\kappa.(\zeta v''.(\zeta v'.(\zeta v.\kappa(\lambda P.v))(\lambda Q.v'))(\forall x.v''))(Px \to Qx) \\
&= \zeta\kappa.(\zeta v'.(\zeta v.\kappa(\lambda P.v))(\lambda Q.v'))(\forall x.Px \to Qx) \\
&= \zeta\kappa.(\zeta v.\kappa(\lambda P.v))(\lambda Q.\forall x.Px \to Qx) \\
&= \zeta\kappa.\kappa(\lambda P.\lambda Q.\forall x.Px \to Qx)
\end{aligned}$$

$$\begin{aligned}
[\![everyone]\!] &= [\![\lambda Q.\forall x(Qx)]\!] \\
&= \zeta\kappa.[\![\forall x(Qx)]\!](\zeta v.\kappa(\lambda Q.v)) \\
&= \zeta\kappa.(\zeta\kappa'.[\![Qx]\!](\zeta v'.\kappa'(\forall x(v'))))(\zeta v.\kappa(\lambda Q.v)) \\
&= \zeta\kappa.(\zeta\kappa'.(\zeta\kappa''.\kappa''(Qx))(\zeta v'.\kappa'(\forall x(v'))))(\zeta v.\kappa(\lambda Q.v)) \\
&= \zeta\kappa.(\zeta\kappa''.\kappa''(Qx))(\zeta v'.(\zeta v.\kappa(\lambda Q.v))(\forall x(v'))) \\
&= \zeta\kappa.(\zeta v'.(\zeta v.\kappa(\lambda Q.v))(\forall x(v')))(Qx) \\
&= \zeta\kappa.(\zeta v.\kappa(\lambda Q.v))(\forall x(Qx))) \\
&= \zeta\kappa.\kappa(\lambda Q.\forall x(Qx))
\end{aligned}$$

$$
\begin{aligned}
[\![\mathit{only}]\!] &= [\![\lambda z.\lambda P.(Pz \wedge \forall x(Px \rightarrow x = z))]\!] \\
&= \zeta\kappa.[\![\lambda P.(Pz \wedge \forall x(Px \rightarrow x = z))]\!](\zeta v.\kappa(\lambda z.v)) \\
&= \zeta\kappa.(\zeta\kappa'.[\![Pz \wedge \forall x(Px \rightarrow x = z)]\!](\zeta v'.\kappa'(\lambda P.v')))(\zeta v.\kappa(\lambda z.v)) \\
&= \zeta\kappa.(\zeta\kappa'.(\zeta\kappa''.\kappa''(Pz \wedge \forall x(Px \rightarrow x = z))(\zeta v'.\kappa'(\lambda P.v')))(\zeta v.\kappa(\lambda z.v)) \\
&= \zeta\kappa.(\zeta\kappa''.\kappa''(Pz \wedge \forall x(Px \rightarrow x = z))(\zeta v'.(\zeta v.\kappa(\lambda z.v))(\lambda P.v')) \\
&= \zeta\kappa.(\zeta v'.(\zeta v.\kappa(\lambda z.v))(\lambda P.v')))(Pz \wedge \forall x(Px \rightarrow x = z)) \\
&= \zeta\kappa.(\zeta v.\kappa(\lambda z.v))(\lambda P.Pz \wedge \forall x(Px \rightarrow x = z))) \\
&= \zeta\kappa.\kappa(\lambda z.\lambda P.Pz \wedge \forall x(Px \rightarrow x = z))
\end{aligned}
$$

These transformations imply that our set of transformation rules does not result in the problem of types that we pointed out in Section 2.3 for a wide range of determiners.

4.4 Inverse Scope as `shift/reset`

In our analysis, inverse scope can be derived without covert movements. Instead, we define the *inverse scope operator* in terms of the `shift` operator.

Definition 14 (Inverse scope operator). *For any meta-lambda term* ϕ : $(e \rightarrow t) \rightarrow t$,

$$[\phi]_{INV} \stackrel{def}{\equiv} \mathit{shift}\ f.\phi(\lambda y.f \triangleleft (\lambda P.Py)) : (e \rightarrow e \rightarrow t) \rightarrow e \rightarrow t$$

For instance, the sentence (8) has a minor reading (called the "wife-reading") in which "every English man" over-scopes "a woman", that is distinct from a major reading (the "Queen-reading").

(8) A woman loves [every English man]$_{INV}$.

The semantic representation of (8), where the inner generalized quantifer "every" is enclosed by the inverse scope operator, is interpreted in the following way.

$$
\begin{aligned}
&[\![(\lambda P.\mathit{some}(\mathit{woman})(P)([\lambda P.\mathit{every}(\mathit{man})(P)]_{INV}\ \mathit{love}))]\!]_c \\
&= \zeta\kappa.[\![\lambda P.\mathit{some}(\mathit{woman})(P)]\!]_c(\zeta s.[\![[\lambda P.\mathit{every}(\mathit{man})(P)]_{INV}]\!]_c(\zeta e.[\![\mathit{love}]\!]_c(\zeta l.\kappa(s(e(l)))))) \\
&= \zeta\kappa.[\![[\lambda P.\mathit{every}(\mathit{man})(P)]_{INV}]\!]_c(\zeta e.\kappa((\lambda P.\mathit{some}(\mathit{woman})(P))(e\ \mathit{love}))) \\
&= \zeta\kappa.[\![\mathit{shift}\ f.(\lambda P.\mathit{every}(\mathit{man})(P))(\lambda y.f \triangleleft (\lambda P.Py))]\!]_c(\zeta e.\kappa((\lambda P.\mathit{some}(\mathit{woman})(P))(e\ \mathit{love}))) \\
&= \zeta\kappa.(\zeta f.[\![(\lambda P.\mathit{every}(\mathit{man})(P))(\lambda y.f \triangleleft (\lambda P.Py))]\!]_c(\zeta x.x))(\zeta e.\kappa((\lambda P.\mathit{some}(\mathit{woman})(P))(e\ \mathit{love}))) \\
&= \zeta\kappa.(\zeta f.\mathit{every}(\mathit{man})(\lambda y.f \triangleleft (\lambda P.Py)))(\zeta e.\kappa((\lambda P.\mathit{some}(\mathit{woman})(P))(e\ \mathit{love}))) \\
&= \zeta\kappa.\mathit{every}(\mathit{man})(\lambda y.\kappa(\mathit{some}(\mathit{woman})(\mathit{love}(y))))
\end{aligned}
$$

This analysis predicts that any quantifier enclosed by an inverse scope operator cannot over-scope the lexical item that *resets*. For example, the inverse scope readings, *every* > *some*, in the sentences (9a) and (10a) are predicted to be available, while they are not in the sentences (9b) and (10b).

(9) a. Some woman introduced Bill to [every man]$_{INV}$.
b. Some woman only introduced [Bill]$_F$ to [every man]$_{INV}$.

(10) a. Some woman introduced [every man]$_{INV}$ to Sue.
b. Some woman only introduced [every man]$_{INV}$ to [Sue]$_F$.

5 Conclusion

In this paper, we have adopted, as a basic framework, meta-lambda calculus, as proposed in Bekki (2009), in which computational monads are represented as a triple of meta-lambda terms, called internal monads. Then we defined an internal monad for delimited continuations, which determines the translation rule of lambda terms, and defined two control operators, called *shift* and *reset*, under this setting.

We also showed that this setting properly serves as a continuation-based theory of formal semantics, and that it is free from the four crucial empirical/ theoretical problems of Continuized Semantics, and we have demonstrated how the use of control operators enables us to compute the meaning of sentences involving phenomena such as "focus movement" and inverse scope, without recourse to "covert movements."

References

Barker, C.: Introducing Continuations. In: Hastings, R., Jackson, B., Zvolenszky, Z. (eds.) SALT 11, CLC Publications, Ithaca (2001)

Barker, C.: Continuations and the Nature of Quantification. Natural Language Semantics 10(3), 211–241 (2002)

Barker, C.: Continuations in Natural Language. In: H. Thielecke (ed.): the Fourth ACM SIGPLAN Continuations Workshop (CW 2004). Technical Report CSR-04-1, School of Computer Science, University of Birmingham, Birmingham B15 2TT. United Kingdom, pp. 1–11 (2004)

Barker, C., Shan, C.-c.: Types as Graphs: Continuations in Type Logical Grammar. Journal of Logic, Language and Information 15(4), 331–370 (2006)

Bekki, D.: Monads and Meta-Lambda Calculus. In: Hattori, H. (ed.) JSAI 2008 Conference and Workshops. LNCS (LNAI), vol. 5447, pp. 193–208. Springer, Heidelberg (2009)

Danvy, O., Filinski, A.: Abstracting Control. In: LFP 1990, the 1990 ACM Conference on Lisp and Functional Programming, pp. 151–160 (1990)

Danvy, O., Filinski, A.: Representing control. Mathematical Structures in Computer Science 2(4) (1992)

de Groote, P.: Type raising, continuations, and classical logic. In: van Rooij, R., Stokhof, M. (eds.) The 13th Amsterdam Colloquium. Institute for Logic, Language and Computation, pp. 97–101. Universiteit van Amsterdam (2001)

Dybvig, R.K., Peyton-Jones, S., Sabry, A.: A Monadic Framework for Delimited Continuations. Journal of Functional Programming 17(6), 687–730 (2007)

Felleisen, M.: The Theory and Practice of First-Class Prompts. In: 15th ACM Symposium on Principles of Programming Languages, pp. 180–190 (1988)

Hayashishita, J.R.: 'Syntactic Scope and Non-Syntactic Scope'. In: Doctoral dissertation, University of Southern California (2003)

Kratzer, A.: Scope or pseudoscope? Are there wide scope indefinites? In: Rothstein, S. (ed.) Events and Grammar, pp. 163–196. Kluwer, Dordrecht (1998)

Otake, R.: Delimited continuation in the grammar of Japanese. Talk presented at Continuation Fest, Tokyo (2008)

Partee, B., Rooth, M.: Generalized conjunction and type ambiguity. In: Bauerle, R., Schwarze, C., Von Stechow, A. (eds.) Meaning, Use and Interpretation of Language, pp. 361–393. Walter De Gruyter Inc., Berlin (1983)
Plotkin, G.D.: Call-by-Name, Call-by Value and the Lambda Calculus. Theoretical Computer Science 1, 125–159 (1975)
Rooth, M.: A Theory of Focus Interpretation. Natural Language Semantics 1, 75–116 (1992)
Shan, C.-c.: A continuation semantics of interrogatives that accounts for Baker's ambiguity. In: Jackson, B. (ed.) Semantics and Linguistic Theory XII, Ithaca, pp. 246–265. Cornell University Press (2002)
Shan, C.-c.: Linguistic side effects. In: Barker, C., Jacobson, P. (eds.) Direct compositionality, pp. 132–163. Oxford University Press, Oxford (2007)
Shan, C.-c., Barker, C.: Explaining Crossover and Superiority as Left-to-right Evaluation. Linguistics and Philosophy 29(1), 91–134 (2006)
Sitaram, D.: Handling Control. In: The ACM Conference on Programming Language Design and Implementation (PLDI 1993). ACM SIGPLAN Notices, vol. 28, pp. 147–155. ACM Press, New York (1993)
Sitaram, D., Felleisen, M.: Control delimiters and their hierarchies. LISP and Symbolic Computation 3(1), 67–99 (1990)
Stratchey, C., Wadsworth, C.: 'Continuations: a mathematical semantics for handling full jumps'. Technical report, Oxford University, Computing Laboratory (1974)
Wagner, M.: NPI-Licensing and Focus Movement. In: Georgala, E., Howell, J. (eds.) SALT XV, CLC Publications, Ithaca (2006)
Winter, Y.: Flexibility Principle in Boolean Semantics: coordination, plurality and scope in natural language. MIT Press, Cambridge (2001)

Appendix

A Language of Meta-Lambda Calculus

A.1 Types and Meta-Types

The syntax of MLC is specified by the following definitions.

Definition 15 (Alphabet for MLC). *An alphabet for MLC is a sextuple* $\langle \mathcal{GT}, \mathcal{C}on, \mathcal{M}con, \mathcal{V}ar, \mathcal{M}var, \mathfrak{S} : \mathcal{M}var \rightarrow \mathcal{P}ow(\mathcal{V}ar)\rangle$*, which respectively represents a finite collection of ground types, constant symbols, meta-constant symbols, variables, meta-variables and an assignment function of free variables for each meta-variable.*

Definition 16 (Types). *The collections of types (notation* $\mathcal{T}yp$*) for an alphabet* $\langle \mathcal{GT}, \mathcal{C}on, \mathcal{M}con, \mathcal{V}ar, \mathcal{M}var, \mathfrak{S}\rangle$ *is defined by the following BNF grammar (where* $\gamma \in \mathcal{GT}$*).*

$$\mathcal{T}yp \ := \ \gamma \mid \mathcal{T}yp \rightarrow \mathcal{T}yp$$

Definition 17 (Context). *A context is a finite list of pairs that are members of* $\mathcal{V}ar \times \mathcal{T}yp$ *(notation* $\Gamma = x_1 : \alpha_1, \ldots, x_n : \alpha_n$*), where* $x_1, \ldots, x_n$ *are distinct variables.*

Definition 18 (Meta-types). *The collection of meta-types (notation $\mathcal{M}type$) for an alphabet $\langle \mathcal{GT}, \mathcal{C}on, \mathcal{M}con, \mathcal{V}ar, \mathcal{M}var, \mathfrak{S} \rangle$ is defined by the following BNF grammar (where Γ is a context and $\tau \in \mathcal{T}yp$).*

$$\mathcal{M}typ \ := \ \Gamma \vdash \tau \mid \mathcal{M}typ \Rightarrow \mathcal{M}typ$$

Definition 19 (Meta-context). *A meta-context is a finite list of pairs that are members of $\mathcal{M}var \times \mathcal{M}Typ$ (notation $\Delta = X_1 : \sigma_1, \ldots, X_n : \sigma_n$), where $X_1, \ldots, X_n$ are distinct meta-variables.*

A.2 Raw-Terms

Definition 20 (Raw-terms). *The collection of raw-terms in Meta-Typed Lambda Calculus (notation Λ) is recursively defined by the following BNF notation, where $x \in \mathcal{V}ar$, $c \in \mathcal{C}on$, $X \in \mathcal{M}var$, and $C \in \mathcal{M}con$.*

$$\begin{aligned} \Lambda ::= & \ x \mid c \mid \lambda x.\Lambda \mid \Lambda\Lambda \\ & \mid X \mid C \mid \zeta X.\Lambda \mid \Lambda \triangleleft \Lambda \end{aligned}$$

A.3 Judgment

Definition 21 (Judgment). *A judgment is a form $\Delta \Vdash M : \sigma$ where Δ is a meta-context, M is a raw-term, and σ is a meta-type, which is derived by the following rules.*

Variable
$$(VAR)\frac{}{\Delta \Vdash x : (\Gamma, x : \alpha, \Gamma' \vdash \alpha)}$$

Constant Symbol
$$(CON)\frac{}{\Delta \Vdash c : (\Gamma \vdash \alpha)} \quad where\ c \in \mathcal{C}on.$$

Lambda Abstraction
$$(LAM)\frac{\Delta \Vdash M : (\Gamma, x : \alpha \vdash \beta)}{\Delta \Vdash \lambda x.M : (\Gamma \vdash \alpha \to \beta)}$$

Functional Application
$$(APP)\frac{\Delta \Vdash M : (\Gamma \vdash \alpha \to \beta) \quad \Delta \Vdash N : (\Gamma \vdash \alpha)}{\Delta \Vdash MN : (\Gamma \vdash \beta)}$$

Substitution
$$(SUB)\frac{\Delta \Vdash M : (\Gamma, x : \alpha \vdash \beta) \quad \Delta \Vdash N : (\Gamma \vdash \alpha)}{\Delta \Vdash M[N/x] : (\Gamma \vdash \beta)}$$

Definition 22 (Judgment for meta-terms). *A judgment is a form $\Delta \Vdash M : \sigma$ where Δ is a meta-context, M is a meta-term, and σ is a meta-type, which is derived by the following rules.*

Meta-Variable
$$(MVAR)\frac{}{\Delta, X : \sigma, \Delta' \Vdash X : \sigma}$$

Meta-Constant Symbol
$$(MCON)\frac{}{\Delta \Vdash C : \sigma} \quad where\ C \in \mathcal{M}con.$$

Meta-Lambda Abstraction $$(MLAM)\frac{\Delta, X:\sigma \Vdash M:\tau}{\Delta \Vdash \zeta X.M:\sigma \Rightarrow \tau}$$

Meta-Functional Application $$(MAPP)\frac{\Delta \Vdash M:\sigma \Rightarrow \tau \quad \Delta \Vdash N:\sigma}{\Delta \Vdash M \lhd N:\tau}$$

Meta-Substitution $$(MSUB)\frac{\Delta, X:\sigma \Vdash M:\tau \quad \Delta \Vdash N:\sigma}{\Delta \Vdash M[N/X]:\tau}$$

A.4 Free Variables and Free Meta-Variables

Definition 23 (Free Variables and Meta-variables). *The set of free variables and free meta-variables are defined respectively by the following sets of rules.*

$$\begin{aligned} fv(x) &= \{x\} & fmv(x) &= \{\} \\ fv(c) &= \{\} & fmv(c) &= \{\} \\ fv(X) &= \mathfrak{S}(X) & fmv(\lambda x.M) &= fmv(M) \\ fv(\lambda x.M) &= fv(M) - \{x\} & fmv(MN) &= fmv(M) \cup fmv(N) \\ fv(MN) &= fv(M) \cup fv(N) & fmv(X) &= \{X\} \\ fv(\zeta X.M) &= fv(M) & fmv(\zeta X.M) &= fmv(M) - \{X\} \\ fv(M \lhd N) &= fv(M) & fmv(M \lhd N) &= fmv(M) \cup fmv(N) \end{aligned}$$

A.5 Substitution

Definition 24 (Substitution Rules for variables)

$$\begin{aligned} x[L/x] &\stackrel{def}{\equiv} L \\ y[L/x] &\stackrel{def}{\equiv} y \quad where\ y \not\equiv x. \\ c[L/x] &\stackrel{def}{\equiv} c \quad where\ c \in \mathcal{C}on. \\ (\lambda x.M)[L/x] &\stackrel{def}{\equiv} \lambda x.M \\ (\lambda y.M)[L/x] &\stackrel{def}{\equiv} \lambda y.(M[L/x]) \quad where\ x \notin fv(M) \vee y \notin fv(L). \\ (\lambda y.M)[L/x] &\stackrel{def}{\equiv} \lambda w.(M[w/y])[L/x] \quad where\ x \in fv(M) \wedge y \in fv(L). \\ (MN)[L/x] &\stackrel{def}{\equiv} (M[L/x])(N[L/x]) \\ C[L/x] &\stackrel{def}{\equiv} C \quad where\ C \in \mathcal{M}con. \\ (\zeta X.M)[L/x] &\stackrel{def}{\equiv} \zeta X.(M[L/x]) \\ (M \lhd N)[L/x] &\stackrel{def}{\equiv} (M[L/x]) \lhd (N[L/x]) \end{aligned}$$

Remark 25. $X[L/x]$ where $x \in fv(X)$ should be treated independently.

Definition 26 (Substitution Rules for meta-variables)

$$
\begin{aligned}
x[L/X] &\stackrel{def}{\equiv} x \\
c[L/X] &\stackrel{def}{\equiv} c \quad where\ c \in \mathcal{C}on. \\
(\lambda x.M)[L/X] &\stackrel{def}{\equiv} \lambda x.(M[L/X]) \\
(MN)[L/X] &\stackrel{def}{\equiv} (M[L/X])(N[L/X]) \\
X[L/X] &\stackrel{def}{\equiv} L \\
Y[L/X] &\stackrel{def}{\equiv} Y \quad where\ Y \not\equiv X. \\
C[L/X] &\stackrel{def}{\equiv} C \quad where\ C \in \mathcal{M}con. \\
(\zeta X.M)[L/X] &\stackrel{def}{\equiv} \zeta X.M \\
(\zeta Y.M)[L/X] &\stackrel{def}{\equiv} \zeta Y.(M[L/X]) \quad where\ X \notin fmv(M) \vee Y \notin fmv(L). \\
(\zeta Y.M)[L/X] &\stackrel{def}{\equiv} \zeta W.(M[W/Y])[L/X] \quad where\ X \in fmv(M) \wedge Y \in fmv(L). \\
(M \lhd N)[L/X] &\stackrel{def}{\equiv} (M[L/X])(N[L/X])
\end{aligned}
$$

B ζ-Theory

Axiom 27 (Permutation and Meta-Permutation)

$$\frac{\Delta \Vdash M = N : (\Gamma, x : \alpha, y : \beta, \Gamma' \vdash \delta)}{\Delta \Vdash M = N : (\Gamma, y : \beta, x : \alpha, \Gamma' \vdash \delta)} \qquad \frac{\Delta, X : \nu, Y : \upsilon, \Delta' \Vdash M = N : \sigma}{\Delta, Y : \upsilon, X : \nu, \Delta' \Vdash M = N : \sigma}$$

Axiom 28 (Weakening and Meta-Weakening)

$$\frac{\Delta \Vdash M = N : (\Gamma \vdash \beta)}{\Delta \Vdash M = N : (\Gamma, x : \alpha \vdash \beta)} \qquad \frac{\Delta \Vdash M = N : \sigma}{\Delta, X : \nu \Vdash M = N : \sigma}$$

Axiom 29 (Equivalence)

$$(=R)\frac{\Delta \Vdash M : \sigma}{\Delta \Vdash M = M : \sigma} \qquad (=S)\frac{\Delta \Vdash M = N : \sigma}{\Delta \Vdash N = M : \sigma}$$

$$(=T)\frac{\Delta \Vdash L = M : \sigma \quad \Delta \Vdash M = N : \sigma}{\Delta \Vdash L = N : \sigma}$$

Axiom 30 (Replacement)

$$(=\lambda)\frac{\Delta \Vdash M = N : (\Gamma, x : \alpha \vdash \beta)}{\Delta \Vdash \lambda x : \alpha.M = \lambda x : \alpha.N : (\Gamma \vdash \alpha \to \beta)}$$

$$(=F)\frac{\Delta \Vdash M = N : (\Gamma \vdash \alpha)}{\Delta \Vdash F(M) = F(N) : (\Gamma \vdash \beta)} \qquad (=A)\frac{\Delta \Vdash F = G : (\Gamma \vdash \alpha \to \beta)}{\Delta \Vdash F(M) = G(M) : (\Gamma \vdash \beta)}$$

Axiom 31 (Meta-Replacement)

$$(=M\lambda)\frac{\Delta, X:\sigma \Vdash M = N : \tau}{\Delta \Vdash \zeta X:\sigma.M = \zeta X:\sigma.N : \sigma \Rightarrow \tau}$$

$$(=MF)\frac{\Delta \Vdash M = N : \sigma}{\Delta \Vdash F \lhd M = F \lhd N : \tau} \qquad (=MA)\frac{\Delta \Vdash F = G : \sigma \Rightarrow \tau}{\Delta \Vdash F \lhd M = G \lhd M : \tau}$$

Axiom 32 (Function Equations)

$$(\alpha)\frac{\Delta \Vdash M : (\Gamma, x:\alpha \vdash \beta)}{\Delta \Vdash \lambda x.M = \lambda y.M[y/x] : (\Gamma \vdash \alpha \to \beta)} \quad \textit{where } y \notin \textit{fv}(M).$$

$$(\beta)\frac{\Delta \Vdash F : (\Gamma, x:\alpha \vdash \beta) \qquad \Delta \Vdash M : (\Gamma \vdash \alpha)}{\Delta \Vdash (\lambda x:\alpha.F)M = F[M/x] : (\Gamma \vdash \beta)}$$

$$(\eta)\frac{\Delta \Vdash M : (\Gamma \vdash \alpha \to \beta)}{\Delta \Vdash \lambda x:\alpha.(Mx) = M : (\Gamma \vdash \alpha \to \beta)} \quad \textit{where } x \notin \textit{fv}(M).$$

Axiom 33 (Meta-Function Equations)

$$(A)\frac{\Delta, X:\sigma \Vdash M : \tau}{\Delta \Vdash \zeta X.M = \zeta Y.M[Y/X] : \sigma \Rightarrow \tau} \quad \textit{where } y \notin \textit{fmv}(M).$$

$$(B)\frac{\Delta, X:\sigma \Vdash F : \tau \qquad \Delta \Vdash M : \sigma}{\Delta \Vdash (\zeta X:\sigma.F) \lhd M = F[M/X] : \tau}$$

$$(H)\frac{\Delta \Vdash M : \sigma \Rightarrow \tau}{\Delta \Vdash \zeta X:\sigma.(MX) = M : \sigma \Rightarrow \tau} \quad \textit{where } X \notin \textit{fmv}(M).$$