1

Only Control Effects and Dependent Types

YOUYOU CONG, Ochanomizu University
WILLIAM J. BOWMAN, Northeastern University

ACM Reference format:
Youyou Cong and William J. Bowman. 2016. Only Control Effects and Dependent Types. 1, 1, Article 1 (January 2016), 3 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

In natural language semantics, control operators, like **shift** and **reset** (Danvy and Filinski 1990), have been used to solve a major challenge in formalizing *compositional* semantics for natural languages—how to represent sentences that manipulate *scope* (Barker 2004). Natural language semantics is concerned with how to represent the meaning of natural language sentences. For example, we could represent the sentence "John loves Mary" as the logical predicate $\mathbf{Love}\,(j,\ m)$. Such semantic representations should be built *compositionally*, i.e., the meaning of the whole sentence is computed from the meanings of constituent words and the rules used to combine them. This principle is intuitive, and explains the reason why we are able to interpret sentences that we have never encountered before. However, compositional calculation of a semantic representation poses difficulties when the sentence contains phrases that *take scope*, which can be understood as that of quantifiers in predicate logic. *Scope-taking* is a concept from natural language semantics unrelated to the notion of scope in programming language semantics. An example of scope-taking phrases is the adverb "only" in "John only loves Mary". One way to encode this sentence is as $\forall x.\mathbf{Love}\,(j,\ x) \leftrightarrow x = m$.

Dependent types have been used to solve another major challenge in natural language semantics—modeling *anaphoric phrases*, *i.e.*, phrases that require the information of previous sentences such as "he" in the discourse "Someone entered. He whistled." Bekki and Mineshima (2017) develop Dependent Type Semantics (DTS), which uses dependent types to represent the meaning of sentences, and shows an elegant solution to handling anaphoric phrases. For example, the first sentence of the above discourse is encoded as the type $\Sigma\, x : \mathbf{Entity}\,.\,\mathbf{Enter}\,(x)$, and the second sentence as $\mathbf{Whistle}\,(\mathbf{fst}\ p)$, where p is a proof of the first sentence, *i.e.*, a term of type $\Sigma\, x : \mathbf{Entity}\,.\,\mathbf{Enter}\,(x)$.

Unfortunately, in programming language semantics, combining control operators and dependent types is an open problem. For example, the continuation-passing style (CPS) translation, which is often used to implement control operators, results in ill-typed terms if the language includes inductive types with dependent eliminations (Barthe and Uustalu 2002). Further work shows that introducing **call/cc** to a system with Σ types and equality leads to inconsistency (Herbelin 2005). Recent work by Herbelin (2012) defines a fragment of dependent type theory that can safely use control operators—the *negative-elimination-free* fragment.

In this work, we model a dependently typed language with the **shift** and **reset** operators. The syntax of our model is given in Figure 1. While our model so far omits Σ types, and thus cannot yet represent anaphoric phrases,

DOI: 10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{rrl}
\textit{Universes} & U & ::= \mathit{Type}\ i \mid \Pi\, x : X \,.\, U \mid \Pi\, \alpha : U_1 \,.\, U_2 \\
\textit{Types} & A, X & ::= \alpha \mid c \mid \lambda\, x : X \,.\, A \mid \lambda\, \alpha : U \,.\, A \mid A\ e \mid A\ B \mid \Pi\, x : X \,.\, Y \mid \Pi\, \alpha : U \,.\, X \mid \textbf{shift}\ A \mid \textbf{reset}\ A \\
\textit{Terms} & e & ::= x \mid c \mid \lambda\, x : X \,.\, e \mid \lambda\, \alpha : U \,.\, e \mid e_1\ e_2 \mid e\ A \mid \textbf{shift}\ A
\end{array}
$$

Fig. 1. Syntax of the source language

it yields interesting insights into the problem of control operators in dependent type theory. In the presence of dependent types, there are 8 variants of the **shift** operator. These variants depend on: (i) whether the **shift** appears as a term or a type; (ii) whether it captures a term-level context or a type-level context; and (iii) whether it returns a term or a type.

In natural language semantics, we need two of these **shift** operators. First, we need the *term-to-type* **shift**, which is a term that captures a type-level context and returns a type. This can represent the sentence "John only loves *Mary*". In this sentence, we place the emphasis, or the *focus* of "only", on the phase "*Mary*". This sentence means that the only person John loves is Mary. This is the same interpretation of the sentence we gave earlier; we will see a second interpretation shortly. In DTS, the intended meaning of this sentence is represented as $\Pi\, x : \mathit{Entity} \,.\, \textbf{Love}\,(j,\ x) \leftrightarrow x = m$, where j and m are terms of type **Entity** and $\textbf{Love}\,(j,\ x)$ is a type that depends on entities j and x. Loosely speaking, we can obtain this representation by translating the sentence into the formula $\textbf{Love}\,(j,\ \textbf{Only}\,(m))$. We implement $\textbf{Only}\,()$ using **shift** to capture the context, and by placing **reset** at the start of the sentence. The expression $\textbf{Only}\,(m)$ receives the context $\textbf{Love}\,(j,\ [\,.\,])$. This is a type-level context, but the argument position $[\,.\,]$ is a term—hence this is a term-to-type **shift**. We then replace the context, adding a quantifier and the additional constraint to implement the meaning of the adverb "only".

$$\textbf{Only}\,(f) = \textbf{shift}\,(\lambda\, k : (\Pi\, \_ : F \,.\, \mathit{Type}\ 0) \,.\, \Pi\, x : F \,.\, k\ x \leftrightarrow x = f)$$

Here, f is the focused phrase and F is the type of f. Using this, we build the formula $\textbf{reset}\,(\textbf{Love}\,(j,\ \textbf{Only}\,(m)))$, which evaluates to the desired representation.

Second, we need the *type-to-type* **shift**, which is a type that captures a type-level context and returns a type. This can represent the sentence "John only *loves* Mary". Unlike before, we place the emphasis on "loves" instead of on "Mary", yielding a different interpretation of the sentence. This sentence means the only thing John does with Mary is love her. We use the type-to-type **shift** in the position of "loves". Note that "loves" is a type-level function, and capture the continuation "John [.] Mary", which is a type-level context.

$$
\begin{aligned}
&\textbf{reset}\ ((\textbf{shift}\ (\lambda\, k : (\Pi\, \_ : (\Pi\, x : \textbf{Entity} \,.\, \Pi\, y : \textbf{Entity} \,.\, \mathit{Type}\ 0) \,.\, \mathit{Type}\ 0) \,.\\
&\qquad\qquad \Pi\, p : (\Pi\, x : \textbf{Entity} \,.\, \Pi\, y : \textbf{Entity} \,.\, \mathit{Type}\ 0) \,.\, k\ p \leftrightarrow p = \textbf{Love}))\ m\ j)
\end{aligned}
$$

We give a prototype implementation of *term-to-type* and *type-to-type* **shift** operator in Cur (Bowman 2016), a dependently typed language with support for safe and sophisticated user-defined extensions. The implementation is a type-preserving call-by-value CPS translation into the core language of Cur, which is similar to the Calculus of Inductive Constructions. The extensions are guaranteed to be sound. As long as the **shift** and **reset** extensions are used over the negative-elimination-free fragment, the extensions will be well-typed. Otherwise, the translation is not type-preserving and the result of the translation does not type-check in Cur. Our goal is to extend the model and the implementation with Σ types to support modeling compositional natural language semantics.

We propose a 20-minute talk, in which we briefly introduce natural language semantics, present the semantics of our model, and give a brief demo of our implementation. In particular we want to communicate the different variants of **shift** operators that arise in dependent type theory. We use natural language semantics as a motivating example, but we hope to find other useful applications from making these variants explicit. We will conclude with the challenges of supporting more sentences, in particular those with anaphoric phrases that rely on Σ types.

REFERENCES

Chris Barker. 2004. Continuations in Natural Language. *Continuation Workshop* 4 (2004), 1–11.

Gilles Barthe and Tarmo Uustalu. 2002. CPS Translating Inductive and Coinductive Types. *ACM SIGPLAN Notices* 37, 3 (2002), 131–142. https://doi.org/10.1145/509799.503043

Daisuke Bekki and Koji Mineshima. 2017. Context-passing and Underspecification in Dependent Type Semantics. In *Studies in Linguistics and Philosophy*. Springer, 11–41. https://doi.org/10.1007/978-3-319-50422-3_2

William J. Bowman. 2016. Growing a Proof Assistant. In *Higher-Order Programming with Effects*. https://williamjbowman.com/papers/#cur

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming (LFP)*. ACM, ACM Press, 151–160. https://doi.org/10.1145/91556.91622

Hugo Herbelin. 2005. On the Degeneracy of Σ-types in Presence of Computational Classical Logic. In *International Conference on Typed Lambda Calculi and Applications (TLCA'05)*. Springer-Verlag, Berlin, Heidelberg, 209–220. https://doi.org/10.1007/11417170_16

Hugo Herbelin. 2012. A Constructive Proof of Dependent Choice, Compatible with Classical Logic. In *Symposium on Logic in Computer Science*. IEEE Computer Society, 365–374. https://doi.org/10.1109/lics.2012.47