Algebraic Effects and Handlers in Natural Language Interpretation

Jiří Maršík[1] Maxime Amblard[1]

[1]LORIA, UMR 7503, Université de Lorraine, CNRS, Inria, Campus Scientifique, F-54506 Vandœuvre-lès-Nancy, France
{jiri.marsik, maxime.amblard}@loria.fr

Abstract

Phenomena on the syntax-semantics interface of natural languages have been observed to have links with programming language semantics, namely computational effects and evaluation order. We explore this connection to be able to profit from recent development in the study of effects. We propose adopting algebraic effects and handlers as tools for facilitating a uniform and integrated treatment of different non-compositional phenomena on the syntax-semantics interface.

In this paper, we give an exposition of the framework of algebraic effects and handlers with an eye towards its applicability in computational semantics. We then present some exemplary analyses in the framework: we study the interplay of anaphora and quantification by translating the continuation-based dynamic logic of de Groote into a more DRT-like theory and we propose a treatment of overt wh-movement which avoids higher-order types in the syntax.

1 Introduction

In the formal study of the syntax-semantics interface, researchers try to discover a systematic translation from the syntactic structures of utterances to their denotations. This translation is often performed indirectly by translating the syntactic structures into a meta-language of semantic representations. We will be studying the challenges of translating syntactic structures to formulas of Church's higher-order logic.

A systematic account of the syntax-semantics interface should be compositional, i.e. the denotations of complex utterances should be functions of the utterances' constituents and their manners of composition. In the style of abstract categorial grammars, we take our syntactic structures to be λ-terms and we demand that the translation from syntax to semantics be a homomorphism. This means that we view the syntactic structure as a program and our goal is to find suitable definitions/interpretations for the constructs of the language this program is written in. To describe the process of building up and gluing together the semantic representation, we will use a meta-language also. This language is often the same as the language of the semantic representations, i.e. λ-calculus, which makes the boundary between the two quite blurry.

Work in this paradigm has focused on phenomena that seem to defy the widely accepted principle of compositionality. We will consider several examples of such work now by looking at the case of a transitive verb. We will focus on in-situ quantification and anaphora. However, similar phenomena could also have been demonstrated on treatments of implicit arguments [Blom et al., 2012] or events [Qian and Amblard, 2011].

(1) Mary read every book.
$\forall x.\mathbf{book}(x) \to \mathbf{read}(\mathbf{Mary}, x)$

To handle in-situ quantification, the verb is traditionally assigned the denotation[1]:

$$\begin{aligned}[\![\textsc{read}]\!] &: ((\iota \to o) \to o) \to ((\iota \to o) \to o) \to o \\ [\![\textsc{read}]\!] &= \lambda so.s(\lambda x.o(\lambda y.\mathbf{read}(x, y)))\end{aligned}$$

One way to look at this is to say that NPs denote generalized quantifiers (type $(\iota \to o) \to o$) and that transitive verbs are relations on generalized quantifiers. We can also think of this as introducing control effects (continuations) into our glue language [Barker, 2002]. NPs can have non-local effects on the construction of the semantic representation by taking scope over their continuations. $[\![\textsc{read}]\!]$ is the result of (partially) CPS-transforming the simple $\lambda so.\mathbf{read}(s, o)$.

(2) Mary$_1$ read her$_1$ favorite book.
$\mathbf{read}(\mathbf{Mary}, \mathbf{favorite\text{-}book}(\mathbf{Mary}))$

In order to interpret (2), we need to link the antecedent with the anaphoric pronoun since the semantic representation of the latter is dependent on that of the former. In accordance with dynamic semantics [Kamp and Reyle, 1993], we can analyze this by positing a store into which discourse referents are introduced and from which they are later retrieved. This boils down to extending our glue language with state. This is the strategy employed in the continuation-based dynamic logic of de Groote [2006]. The type of NP denotations becomes in turn more complex to reflect the fact that NPs can access the current state and manipulate their continuations:

$$[\![np]\!] = (\iota \to \gamma \to (\gamma \to o) \to o) \to \gamma \to (\gamma \to o) \to o$$

The first argument (type $\iota \to \gamma \to (\gamma \to o) \to o$) corresponds to a continuation delimited by the containing tensed clause, the second argument (type γ) is the context, where we find e.g. the available discourse referents, and the third argument (type $\gamma \to o$) corresponds to an open-ended discourse-wide continuation. γ lets us access anaphoric state while the two different continuations serve to enforce DRT accessibility constraints [Kamp and Reyle, 1993] (e.g. the universal quantifier in *everyone* provides a referent for and takes scope over only its containing clause while the existential quantifier in *someone* provides a referent for and scopes over even following clauses).

The denotation this theory assigns to the transitive verb is:[2]

[1]As in Church's Simple Type Theory, we use ι for the type of individuals and o for the type of propositions.

[2]$\bar{o}$, a dynamic proposition, is shorthand for $\gamma \to (\gamma \to o) \to o$, where γ is the anaphoric state, i.e. the referent store, and $\gamma \to o$ is the continuation.

$$
\begin{aligned}
&[\![\textsc{read}]\!] : [\![np]\!] \to [\![np]\!] \to [\![s]\!] \\
&[\![\textsc{read}]\!] : ((\iota \to \overline{o}) \to \overline{o}) \to ((\iota \to \overline{o}) \to \overline{o}) \to \overline{o} \\
&[\![\textsc{read}]\!] : ((\iota \to \gamma \to (\gamma \to o) \to o) \to \gamma \to (\gamma \to o) \to o) \\
&\qquad \to ((\iota \to \gamma \to (\gamma \to o) \to o) \to \gamma \to (\gamma \to o) \to o) \\
&\qquad \to \gamma \to (\gamma \to o) \to o \\
&[\![\textsc{read}]\!] = \lambda so.s(\lambda x.o(\lambda ye\phi.\mathbf{read}(x,y) \land \phi e))
\end{aligned}
$$

We interpret the theory several ways. One can say that READ is a dynamic relation ($\alpha \to \beta \to \overline{o}$) on generalized dynamic quantifiers (type $(\iota \to \overline{o}) \to \overline{o}$). We also hold another view: instead of considering the dynamic proposition $\overline{o}$ as a semantic representation, we see it as an effectful computation[3] that accesses some anaphoric state and manipulates continuations to produce a term of type o, the final semantic representation.

2 Computation on the Syntax-Semantics Frontier

In the preceding examples, we have seen that seemingly non-compositional phenomena can be accounted for by admitting some sort of effect into our glue language. If we go back to the metaphor of our syntactic structures being programs that evaluate to their semantic representations, it seems that the language these programs are written in exhibits a lot of effects. In the previous chapters, we have seen state and continuations, but a look at implicit arguments and events would lead us to also consider partiality and environment-dependence, respectively.

Our chief motivation is to unite the treatments presented in the previous section and to start formalizing the interactions between the phenomena treated therein. Since effects are an intensely-studied topic in formal semantics of programming languages, it would be convenient to employ a glue language equipped with some notion of effects.

Choosing a language with some fixed set of side effects (such as some general-purpose programming language) would not be practical since as we have seen, treatments of newly discovered phenomena often call on new effects. We would therefore prefer a framework that allows us to abstract over the different effects in our glue language.

One such framework are the notions of computation of Moggi [1991], rendered as monads in category theory and used heavily in popular functional programming. Shan [2002] already examined the potential of a monadic framework for studying the various effects he observed in existing linguistic analyses. However, in his study, he points out that combining different monads is a difficult problem which still lacks a satisfying solution (see references in Shan [2002] and Kammar et al. [2013]).

A compelling alternative to the use of monads for describing effects has gained popularity recently (Bauer and Pretnar [2012], Kammar et al. [2013], Kiselyov et al. [2013]). This new approach is rooted in the algebraic study of effects and handlers by Plotkin and

[3]In a sense similar to the notions of computation of Moggi [1991], popularized as the monads of functional programming.

Pretnar [2009] that builds on the notion of Lawvere theory, which provides a category-theoretical take on universal algebra different from the one offered by monads [Hyland and Power, 2007]. We argue that algebraic effects can solve some of our problems in combining multiple effects in a single glue language and that the dual notion, handlers, can be identified in existing linguistic treatments and is a natural fit for our task.

3 The Case for Effects and Handlers

In the algebraic effects framework, we use a calculus with abstract effectful operations as language primitives [Kammar et al., 2013].

$$\frac{(\mathbf{op} : A \to B) \in E \qquad \Gamma \vdash V : A \qquad \Gamma, x : B \vdash_E M : C}{\Gamma \vdash_E \mathbf{op}\ V\ (\lambda x.M) : C}$$

We take *op* to be an abstract operation with argument type A and result type B. We can invoke *op* by providing it with an argument of the correct type and a continuation which will receive the result of the operation and carry out the rest of the computation.

What is of note in the above judgments is the E subscript. Typing judgments for computations not only serve to prove that a computation will finish by yielding a value of some type (given after the colon), but they also guarantee us that the computation will restrict itself to effectful operations contained in the set E.

Such a type system should be familiar to anyone who has ever used Java. As goes in the Java language specification [Gosling, 2000], "Methods declare the checked exceptions that can arise from their execution, which allows compile-time checking to ensure that exceptional conditions are handled". We can see the invocation of an operation as throwing a (checked) exception whose message contains the argument value and the continuation to be called with the result.

The analogy of operations as exceptions will help us understand the dual notion of a handler. Handlers behave exactly like exception handlers. Exception handlers intercept specific types of exceptions to define how they should be resolved, making use of any content stored in the exception message. General handlers intercept specific abstract operations to define how they should be executed, making use of the supplied argument and the callback continuation. Since general handlers have access to the continuation, they can resume the computation at the point where the abstract operation was invoked, whereas exception handlers generally do not and thus discard the failed computation.[4]

3.1 Denotations for Effects

Let us now look at what would be a suitable denotational semantics for a language with algebraic effects and handlers.

An expression $\Gamma \vdash_E M : C$ can be seen as something that will either compute directly to the final value of type C or something that will invoke one of the operations in E,

[4]The Common Lisp condition and restart system also allows one to resume the computation when handling an exception. However, the continuation available to the handler is not a first-class function and cannot be invoked multiple times.

providing the operation's argument and a continuation.[5] Following is a formula for the type of denotation given to a computation of type C exhibiting effects E. [Bauer and Pretnar, 2012] [Kiselyov et al., 2013]

$$[\![C_E]\!] = C + \sum_{(\mathbf{op}:A \to B) \in E} A \times [\![C_E]\!]^B$$

We can start appreciating some enticing features of this system in comparison to the prevalent monadic approach.

First of all, the types of computations are decomposed into two components: the type of the value being computed and the set of effects the computation can have. This is in contrast with the monadic approach, where both the result type and the effects meet in a single type, with the monad functor being applied to the result type.

As an example, let us take a computation of result type α that accesses state of type γ. In the effects framework, this computation would have type $\alpha_{\{get:1\to\gamma, put:\gamma\to 1\}}$ where *get* and *put* are the abstract operations used to read from and write to the state. In the monadic framework, this computation would be rendered as $State_\gamma(\alpha)$ which is equal to $\gamma \to \alpha \times \gamma$.

If we then consider extending this computation to include, e.g., exceptions, we would get the following developments. In the effects framework, we have $\alpha_{\{get:1\to\gamma, put:\gamma\to 1, raise:\epsilon\to 0\}}$. In the monadic framework, we have $Exc_\epsilon(State_\gamma(\alpha))$, which is $(\gamma \to \alpha \times \gamma) + \epsilon$, or maybe $State_\gamma(Exc_\epsilon(\alpha))$, which is $\gamma \to (\alpha + \epsilon) \times \gamma$.

If we look at the type of the denotation of the computation in the effects framework, we see that the different effects sit side-by-side in an unordered flat collection. The computation is represented as either a value or a request to perform one of the effectful operations. Any interactions between these effects are up to the handlers which will interpret this computation.

On the other hand, in the case of the monadic framework, we are forced to commit from the start to a specific interaction between the two effects. The representation of the computation is then bound to this specific interpretation. Furthermore, the two modes of combining the two monadic effects shown above are not exhaustive. Kiselyov et al. [2013] show an example computation which combines exceptions and non-determinism and that requires the use of the type $Exc_\epsilon(List(Exc_\epsilon(\alpha)))$.

The structure of the denotations in the effects framework makes it easy to embed less expressive computations into contexts that employ a wider palette of effects and to be polymorphic w.r.t. the effects. If we take the example of a stateful computation in a context that also permits exceptions, then we can benefit from the fact that $[\![\alpha_{\{get,put\}}]\!]$ is a subset of $[\![\alpha_{\{get,put,raise\}}]\!]$ (modulo some trivial injection). This means we can take expressions and denotations from linguistic treatments that ignore some linguistic effect and we can plug them directly into our richer integrated treatment without having to do any adaptation, lifting or conversion.

More interestingly, we can also do the converse thanks to effect polymorphism. We can replace any subterm N_1 of some term M with another subterm N_2 which has the

[5]The idea of seeing a computation as denoting either a value or an effect dates back to at least Cartwright and Felleisen [1994].

same type but that also triggers some new unhandled effects E. The resulting term M' will be still well-typed and the unhandled effects E of N_2 will simply propagate to become the unhandled effects of M'. This means that if we have a constituent below which we need to introduce some effect and above which the effect will be handled, we are not obliged to modify the denotation of the intervening constituent. If it has no interaction with the effect, it can remain agnostic. This is in contrast with the manual style of passing around all the contextual arguments, states and continuations.

3.2 Handlers

Now we examine the notion of a handler. If we look back to the definition of $[\![C_E]\!]$ in 3.1, we see that a computation is either a value or an effect. We can think of values as constants and of effects as algebraic operations. More precisely, an effect $op : A \to B$ can be seen as a family of algebraic operations op_a, one for each value a of type A. Each of these operations op_a is a $|B|$-ary operation on computations, combining the computations for all the possible outcomes of the effect into a single computation.

To demonstrate on an example, let us consider an operation for writing strings to some output channel, $print : \sigma \to 1$. This can be thought of as a family of unary operations $print_s$ for every string s. The meaning of $print_{hello}(C)$ is then a computation that first prints *hello* to the output channel and then carries out the computation C. The operator $or : 1 \to 2$ for non-deterministic choice corresponds to a binary operation on computations such that $or(C_1, C_2)$ is a non-deterministic computation that continues by either evaluating C_1 or C_2. This formulation in terms of algebraic operations on computations is what gave this framework the name "algebraic effects".

In this algebraic view, a handler can be seen as a homomorphism which maps the computation it handles to another computation by instantiating certain abstract operations within [Plotkin and Pretnar, 2009] [Pretnar, 2010].

Thinking of handlers as scoped interpretations is instructive. Denotations can be written using environment-dependent abstract operations such as "introduce a new discourse referent (in the current DRS)", "access available discourse referents", "access event under discussion", "access salient world", "scope over the current tensed clause"...

Handlers can then be placed at the appropriate junctions in the syntactic tree to give meaning to these abstract operations. To give several examples: lexical items that necessitate a fresh DRS (negation, universal quantification) will carry handlers for introducing and accessing discourse referents; tensed clauses will use handlers to enforce quantifier scope islands; access to the current event might be handled at the edge of a scope domain; and modality operators would handle references to the salient world.

4 Applications of Effects and Handlers to Computational Semantics

Our first inspiration for adopting effects was to bridge the hierarchical structures of DRSs and the compositional treatment of anaphora in the continuation-based dynamic logic of de Groote [2006].

4.1 DRT, Continuations and Effects

In de Groote [2006], and in more detail in Lebedeva [2012], we can see indefinites introducing some existential quantifier and adding the variable bound by the quantifier into the store as a new discourse referent. In our effects framework, we can think of this complex action as an effect, let us call it **fresh**, whose purpose is to introduce a new discourse referent at the current point of discourse and whose implementation will introduce a quantifier someplace and record the referent in some store. This decomposition into an abstract interface (the **fresh** operation which introduces a discourse referent) and specific implementations that will realize it (handlers for **fresh**) is pertinent. It gives us another way to explain how it is possible that the same indefinite expression can sometimes contribute an existential quantifier and sometimes a universal one, as is the case with the famous example of donkey sentences [Kamp and Reyle, 1993].

Besides introducing discourse referents, lexical items in continuation-based dynamic logic also access the sum total of the available discourse referents to be able to resolve anaphora by choosing a salient antecedent from their ranks. We model this feature by an effect, **get**, which lets computations access the current contents of the discourse store.

With these two operations in hand, we can start rewriting de Groote's continuation-based dynamic logic. In short: the dynamic existential quantifier will essentially boil down to a call to **fresh**, dynamic negation will introduce a handler (a new DRS) and dynamic conjunction will be simple conjunction (with arguments evaluated left-to-right).

$$\begin{aligned}\overline{\exists} &= \lambda P.P(\mathbf{fresh}\ ())\\ \overline{\neg} &= \lambda P_t.\{\neg(\texttt{with}\ drs\ (\mathbf{get}\ ())\ \texttt{handle}\ P_t!)\}\\ \overline{\wedge} &= \lambda P_t Q_t.\{P_t! \wedge Q_t!\}\end{aligned}$$

The fragments we show are faithful excerpts from our experimental grammar. The language used is *Eff* [Bauer and Pretnar, 2012]. As we are dealing with effects, the order of evaluation matters and since *Eff* is call-by-value, we use thunks to simulate call-by-name. For those, we adopt the notation used in Kammar et al. [2013]. $\{M\}$ is short for $\lambda().M$ (i.e. a dummy abstraction serving to build a thunk) and $M!$ is short for $M\ ()$ (i.e. applying a thunk to the dummy unit argument). Furthermore, we mark variables that hold thunks with the subscript t. The exposition could be cleared up by devising a language which separates strictness from abstraction, in much the same way as was already proposed by Kiselyov [2008].

We can now move the dynamicity of continuation-based dynamic logic into the effects, and so instead of having the type $(\iota \to \gamma \to (\gamma \to o) \to o) \to \gamma \to (\gamma \to o) \to o$ for NP denotations, we get $(\iota \to o_E) \to o_E$ for some set of effects E containing **fresh** and **get**.

$$\begin{aligned}[\![\textsc{she}]\!] &= \lambda k.k(sel'_{she}(\mathbf{get}\ ()))\\ [\![\textsc{something}]\!] &= \lambda k.\overline{\exists} x.(k\ x) = \lambda k.k(\mathbf{fresh}\ ())\\ [\![\textsc{every}]\!] &= \lambda nk.\overline{\forall} x.(\{n\ x\} \mathrel{\overline{\to}} \{k\ x\})!\\ [\![\textsc{read}]\!] &= \lambda SO.S(\lambda s.O(\lambda o.\mathbf{read}(s,o)))\end{aligned}$$

We have switched from dynamic propositions ($\overline{o} = \gamma \to (\gamma \to o) \to o$) to effectful computations of static propositions (o_E), as was hinted at in Section 1.

The NP denotations now have type $(\iota \to o_E) \to o_E$. This is still more complicated than the naive type ι one starts with for proper nouns. We can see this type as a computation of ι with access to some continuation of result type o_E. If we look at the denotations, this continuation (the argument k) is properly exploited only in the denotation of *every* where it serves to mark the scope of the quantifier. This scope is supplied to the NP by the tensed clause it is contained in. If we were to take our approach to its logical conclusion, we could turn this into an effect as well. Quantificational noun phrases could employ a **scope_over** effect to introduce some quantifier and tensed clauses could be handlers for these effects, delimiting the scope of these quantifiers.

$$\begin{aligned}
[\![\textsc{she}]\!] &= \{sel'_{she}(\mathbf{get}\ ())\} \\
[\![\textsc{something}]\!] &= \{\mathbf{fresh}\ ()\} \\
[\![\textsc{every}]\!] &= \lambda n.\{\mathbf{scope\_over}\ (\lambda k.\overline{\forall} x.(\{n\ x\} \overline{\Rightarrow} \{k\ x\})!)\} \\
[\![\textsc{read}]\!] &= \lambda s_t o_t.\{\texttt{with}\ \mathit{tensed\_clause}\ \texttt{handle}\ \mathbf{read}(s_t!, o_t!)\}
\end{aligned}$$

We have now arrived at a type for NP denotations that is ι_E, where E is some set of effects including **scope_over**, **get** and **fresh**, the three effects being somewhat analogous to the three arguments that NP denotations abstract over in continuation-based dynamic logic. For now, we have restricted our set of effects that NPs can use to modify the current DRS to just **fresh**, which lets us only introduce new referents. However, indefinite expressions such as *a good book* not only introduce referents but also assert something about them. We can model this by admitting a new effect, **assert**, that will be handled by the same handler as **fresh** and **get**, i.e. by a DRS.

$$\begin{aligned}
[\![\textsc{some}]\!] = \lambda n.\{\ & \texttt{let}\ x = \mathbf{fresh}\ ()\ \texttt{in} \\
& \texttt{let}\ () = \mathbf{assert}\ (n\ x)\ \texttt{in} \\
& x\ \}
\end{aligned}$$

We have arrived at the following system with two kinds of handlers. We have DRSs as handlers that handle three effects that can be seen as the interface of a DRS: we can build up a DRS by introducing new discourse referents (**fresh**) and constraints (**assert**) and we can use the information stored inside to resolve anaphora (**get**). We also have tensed clauses as handlers. They delimit scope islands which define the scopes of the quantifiers contained within (**scope_over**).

4.2 Overt Movement and Effects

Overt movement (as in the case of an object extracted from a relative clause by a relativizer such as *who*) has in the categorial school been often studied as λ-abstraction or hypothetical reasoning.

$$\textsc{who} : (np \multimap s) \multimap n \multimap n$$

Using linear implication for the arrow type lets us enforce that one relativizer can only cover one trace. However, overt wh-movement has more restrictions, especially when it comes to multiple extraction [Pogodalla and Pompigne, 2012].

When we have effects available to us, we can choose an alternative solution. Traces can be seen as inaudible NPs whose semantic representation is computed through an effectful operation, **move**. A relativizer will then handle **move** events in the relative clause. The handler will be a shallow handler [Kammar et al., 2013], meaning it will only handle the first occurrence of the **move** event that will occur in the relative clause, which gives correct predictions in the case of multiple extraction.

$$\begin{aligned} [\![\textsc{who}]\!] &= \lambda r_t n.\lambda x.\ \texttt{let}\ r = \texttt{with}\ \mathit{extract}\ \texttt{handle}\ r_t!\ \texttt{in} \\ &\qquad\qquad (n\ x) \wedge (r\ x) \\ [\![\epsilon]\!] &= \{\mathbf{move}\ ()\} \end{aligned}$$

By making the relativizer handler fail in the case that it does not intercept any **move** effect (i.e. the relative clause contained no free trace), sentences with ungrammatical uses of the relativizer can be ruled out. With this technique, the type of WHO becomes $s \multimap n \multimap n$, which is interesting because using 2nd order types, such as this one, in the syntax makes it possible to use efficient parsing algorithms [Kanazawa, 2007].

5 Conclusion

In this paper, we gave a short exposition to the theory of algebraic effects and handlers and demonstrated how it can be applied to computational semantics. Besides our principal example of dynamic logic and DRT, we also covered extraction. However, that is not the full extent of the applicability of effects and handlers. The event argument under discussion [Qian and Amblard, 2011] and the salient world are both examples of dynamic binding, which is a special case of a handler. Lebedeva [2012] proposes modeling presuppositions and proper names using exceptions, which are a trivial case of handlers. The treatment of implicit arguments in Blom et al. [2012] uses an operator which is also a simple exception handler.

We have not spent much time defending the position that algebraic effects and handlers facilitate the analysis of *multiple* phenomena in the same grammar. However, compelling evidence for using algebraic effects and handlers in contexts that involve multiple computational effects at the same time can be found in existing literature (e.g. Kiselyov et al. [2013], Cartwright and Felleisen [1994]).

For future work, we are interested in developing a simple calculus with effects and a notion of evaluation order suitable for the study of the syntax-semantics interface. Having fixed such a calculus, we would aim to study non-compositional semantic phenomena in concert, not in isolation, and to examine their interactions.

References

Chris Barker. Continuations and the nature of quantification. *Natural language semantics*, 10(3):211–242, 2002.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *arXiv preprint arXiv:1203.1539*, 2012.

Chris Blom, Philippe De Groote, Yoad Winter, and Joost Zwarts. Implicit arguments: event modification or option type categories? In *Logic, Language and Meaning*, pages 240–250. Springer, 2012.

Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software*, pages 244–272. Springer, 1994.

P. de Groote. Towards a montagovian account of dynamics. In *Proceedings of SALT*, volume 16, 2006.

James Gosling. *The Java language specification.* Addison-Wesley Professional, 2000.

Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electronic Notes in Theoretical Computer Science*, 172:437–458, 2007.

Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 145–158. ACM, 2013.

Hans Kamp and Uwe Reyle. *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory.* Number 42. Kluwer Academic Pub, 1993.

Makoto Kanazawa. Parsing and generation as datalog queries. In *Annual Meeting-Association for Computational Linguistics*, volume 45, page 176, 2007.

Oleg Kiselyov. Call-by-name linguistic side effects. In *ESSLLI 2008 Workshop on Symmetric calculi and Ludics for the semantic interpretation*, 2008.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 59–70. ACM, 2013.

Ekaterina Lebedeva. *Expression de la dynamique du discours à l'aide de continuations.* PhD thesis, Université de Lorraine, 2012.

Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93 (1):55–92, 1991.

Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems*, pages 80–94. Springer, 2009.

Sylvain Pogodalla and Florent Pompigne. Controlling extraction in abstract categorial grammars. In *Formal Grammar*, 2012.

Matija Pretnar. Logic and handling of algebraic effects. 2010.

Sai Qian and Maxime Amblard. Event in compositional dynamic semantics. In *Logical Aspects of Computational Linguistics.* 2011.

C. Shan. Monads for natural language semantics. *arXiv preprint cs/0205026*, 2002.