Towards Naturalistic EDSLs using Algebraic Effects

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Abstract

Domain specific programming languages bridge the linguistic and conceptual gap between domain languages and implementation languages. One aspect of bridging the gap is to express the domain specific concepts in a language more natural for the domain experts. In recent years, in linguistics, concepts from computer science such as effect operations (e.g. shift/reset and continuations in general) have successfully been used to provide compositional models for natural language semantics. We propose to pick up the old theme of naturalistic DSLs and reevaluate it in the scope of algebraic effects. Building on the insights of linguists, we demonstrate how linguistic features such as anaphora, quantification and implicature can directly be implemented in the Dotty programming language using a library for algebraic effects. As opposed to ad hoc implementation techniques for naturalistic DSLs, systematically using algebraic effects and *effectful syntax* leads to programs that exactly communicate the usage of linguistic features in their types, offers improved error reporting and better IDE support. We believe that effectful syntax opens up a new interesting perspective on the design and implementation of naturalistic DSLs.

ACM Reference format:
Jonathan Immanuel Brachthäuser. 2017. Towards Naturalistic EDSLs using Algebraic Effects. In *Proceedings of DRAFT, Tübingen, Germany, 2017,* 3 pages.
DOI:

1 Introduction

Bridging the conceptual gap between domain and implementation language by designing domain specific languages (DSLs), which are closer to the natural language, is an idea as old as it is controversial. Natural languages have the reputation of being lexically and syntactically ambiguous, having complicated and context dependent binding structures and often a non-trivial semantics, which rarely is compositional. In short, attributes programmers don't like and programming language designers are at best fascinated by. In consequence, many domain specific languages are still far away from being close to natural language. This is in particular the case for DSLs, which are embedded into a general purpose language. With embedded DSLs the host language additionally imposes its own syntactical restrictions and typing discipline on the DSL designer. While each of those limitations is addressed in its own line of work (e.g. syntax extensions as libraries, domain specific type system extensions) our focus here is on linguistic constructs, which are usually neglected since they are inherently non-context-free.

In about the last decade, many developments in modelling the semantics of natural languages have been inspired by computer science and the theory of abstract machines and control operators in particular. Delimited continuations (using for instance Danvy and Filinski's *shift* and *reset* (1990)) have successfully been used to model quantification (*"John loves every women"*), focus (*"John loves **Mary**"*) and polymorphic coordination (*"John and Mary left"*) (Barker and Shan 2004; Shan 2004, 2005).

Maršík and Amblard (2016) very recently used algebraic effects with handlers to give a compositional semantics to deixis (*"John loves me"*), quantification with scope islands and implicature (*"John, my best friend, loves me"*). Algebraic effects (Plotkin and Power 2003) with handlers (Bauer and Pretnar 2015; Plotkin and Pretnar 2009), described in engineering terms, separate the declaration of effectful operations and their usage from the implementation in effect handlers. Typically, to implement an effect operation the effect handlers obtain access to the delimited continuation, that is, the remaining program after the effect usage up to the handler. Compared to control operators, algebraic effects with handlers closely correspond to multipromt delimited continuations (Kiselyov and Sivaramakrishnan 2016). Roughly, every handler introduces a prompt marker and an effect operation constitutes a shift up to the corresponding handler-prompt. In the context of natural language semantics, shifting to different handlers is essential to enable disambiguation of quantifiers (Barker 2002) and combining different scoping constructs in a single sentence (Maršík and Amblard 2016).

While using insights from linguistics for programming language design is an old idea (Lopes et al. 2003), in this talk proposal, we propose to reconsider the current approach of design and implementation of embedded DSLs by taking recent developments of computer linguistics and algebraic effects into account. We conjecture that algebraic effects are a solid foundation to implement novel syntactic features for embedded naturalistic DSLs, which we refer to as *effectful syntax*.

2 Effectful Syntax – Examples from Natural Language

To provide some context and support our conjecture, this section shows examples from natural language semantics (Maršík and Amblard 2016), implemented in Dotty (the Scala of the future[1]) using a library for algebraic effects (Scala-Effekt[2]). This should serve only as a first example domain. The idea of effectful syntax goes beyond the domain of natural languages and we plan to collaboratively explore other domains with the workshop participants[3]. Our experience in implementing these examples makes us believe that effectful syntax is (i) modular – linguistic effects and handlers can be encapsulated in modules, separated from the remaining syntax (ii) learnable – separating linguistic effects from domain concepts allows the DSL user to learn both separately, also strong (effect) typing enables better error messages for wrong usage of linguistic constructs (iii) maintainable – user programs concisely express

DRAFT, Tübingen, Germany

[1] http://dotty.epfl.ch
[2] http://b-studios.de/scala-effekt
[3] The full code and an interactive programming environment is available online at: https://scastie.scala-lang.org/scfQA0CHQmKfxCKx7TfNgw.

```
trait Sentences[NP, S] {
  def person(name: String): C[NP]
  def man(person: NP): C[S]
  def woman(person: NP): C[S]
  def loves(src: NP, trg: NP): C[S]
  def bestFriendOf(friend: NP, person: NP): C[S]
  def said(person: NP, sentence: S): C[S]
  def forall(f: NP ⇒ C[S]): C[S]
  def and(first: S, second: S): C[S]
}
```

(a) Syntax of the *Sentences* EDSL as a shallow embedding.

```
(effect signature)  trait Speaker extends Eff {def speaker(): Op[NP]}
(effect operation)  def me: NP using Speaker
(effect handler)    def saidQuote(speaker: NP, sentence: S using Speaker): C[S]

(effect signature)  trait Scope extends Eff {def scope[A](k:(A ⇒ C[S]) ⇒ C[S]): Op[A]}
(effect operation)  def every(pred: NP ⇒ C[S]): NP using Scope
(effect handler)    def scoped(f: S using Scope): C[S]

(effect signature)  trait Implicature extends Eff {def imply(s: S): Op[Unit]}
(effect operation)  def whoIs(person: NP, pred: NP ⇒ C[S]): NP using Implicature
(effect handler)    def accommodate(f: S using Implicature): C[S]
```

(b) Effect signatures and handlers for the linguistic effects `Speaker`, `Scope` and `Implicature`.

Figure 1. Syntax of the *Sentences* EDSL and the interface for linguistic effects.

the use of linguistic features in their (effect) types, opening up opportunity for refactorings and IDE support.

Figure 1a defines the syntax of our EDSL for sentences. The type constructor C (pronounced "control") represents effectful computation, and is a monad provided by the Effekt library with the corresponding monadic methods and properties. To also allow *effectful semantics* of the EDSL, all return types are wrapped in C. In the definition of linguistic effects (Figure 1b) and in user programs, we assume S and NP to be the type used for the semantics domain of sentences and nominal phrases, respectively[4].

The Speaker Effect. We begin with a simple sentence that uses the speaker effect to refer to the contextual speaker of the sentence:

```
val s1: S using Speaker = john said {mary loves me}
```

The type annotation of sentence s_1 tells us that the sentence uses the speaker effect[5]. Trying to run the example sentence without providing a speaker will give an error similar to: *"This sentence uses 'me' and requires a speaker to be in the context"*. The speaker effect can be handled locally by using the handler *saidQuote*:

```
val s2: C[S] = john saidQuote {mary loves me}
```

The type of the sentence reflects that no effect is left to be handled, so we can run the sentence to obtain `said(John, loves(Mary, John))`.

The Scope Effect. Passing down context information like we did with the Speaker effect does not yet require algebraic effects with handlers, since we could just as well have used implicits arguments in Scala to pass down the speaker. Things become more interesting when we consider the scope effect, which, similar to a CPS monad, can be used to model universal quantification.

```
val s3: C[S] = scoped {john saidQuote every(woman) loves me}
```

Here, the effect operation *every* takes a predicate and uses the scope effect to generate a universal quantification at the point where the effect is handled by *scoped*. Running s_3, we see that this leads to a systematic "rewrite" of the syntax tree, moving the introduced binder up to the first occurrence of the handler *scoped*.

```
forall(x => implies(woman(x), said(John, loves(x, John))))
```

[4]We also assume variants of operations like *said* that are lifted to C in their arguments. Following a Scala standard pattern for extension methods, binary operations are written infix by attaching a corresponding method to the first argument (e.g. `def said(s: C[S]): C[S]` for the lifted variant).

[5]The Effekt library employs a capability passing style: capabilities are created by effect handlers and passed down to the usage of the effect. Effekt makes use of the recently introduced Dotty feature of *implicit function types* and defines the following type alias `type using[A, E] = implicit Cap[E] ⇒ C[A]`, which can be used infixed in Scala. Thus, the sentence s_1 is equivalent to the more explicit `val s1 = implicit(c: Cap[Speaker]) ⇒ john.said(mary.loves(me(c)))`

The Implicature Effect. The last effect we present, implicature, is similar to the scope effect in that it uses the continuation to achieve a rewriting of the syntax tree.

```
val s4: S using Speaker =
  accommodate {mary loves {john whoIs {_ bestFriendOf me}}}
```

Running the example sentence s_4 with speaker Pete, shows how the annotated apposition is lifted up to the *accommodate* handler and introduces a conjunction.

```
and(bestFriendOf(John, Pete), loves(Mary, John))
```

The sentence s_4 also shows that multiple linguistic effects can naturally be combined. The type of statements expresses which effects are not yet handled. The idea of effectful syntax is of course completely independent of Scala and the sentence DSL could similarly be implemented in other languages that support algebraic effects and handlers. However, since we embedded the DSL into Scala, we could customize the compile time errors, when effects are not handled. This customization might be an interesting feature addition for languages with native support for algebraic effects like Koka (Leijen 2014).

3 Conclusions

In the talk, we will suggest to consider algebraic effects for the design and implementation of more naturalistic EDSLs. We hope that our proposal triggers a vivid discussion about other application domains and sparks new research opportunity to develop novel abstraction mechanisms for effectful syntax, assisting the design and maintenance of naturalistic DSLs.

References

Chris Barker. 2002. Continuations and the nature of quantification. *Natural language semantics* (2002).

Chris Barker and Chung-chieh Shan. 2004. Continuations in natural language. *CW* (2004).

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA.

Oleg Kiselyov and KC Sivaramakrishnan. 2016. Eff directly in OCaml. In *ML Workshop*.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Workshop on Mathematically Structured Functional Programming*.

Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. 2003. Beyond AOP: Toward Naturalistic Programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (Onward! track)*. ACM, Anaheim.

Jirka Maršík and Maxime Amblard. 2016. Introducing a Calculus of Effects and Handlers for Natural Language Semantics. In *International Conference on Formal Grammar*. Springer.

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *European Symposium on Programming*. Springer, 80–94.

Chung-chieh Shan. 2004. Delimited continuations in natural language. In *Continuation Workshop*.

Chung-chieh Shan. 2005. Linguistic Side Effects. In *In Proceedings of the Eighteenth Annual IEEE Symposium on Logic and Computer Science (LICS 2003) Workshop on Logic and Computational*. University Press.