Implementation of static and dynamic semantics for a calculus with algebraic effects and handlers using `PLT Redex`

(Implementacja statycznej i dynamicznej semantyki rachunku z efektami algebraicznymi i ich obsługą z pomocą biblioteki `PLT Redex`)

Maciej Buszka

Praca inżynierska

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

24 stycznia 2019

Abstract

English abstract

Abstrakt w języku polskim

Contents

Chapter 1

Introduction

Algebraic effects [8] are an increasingly popular technique of structuring computational effects. They allow for seamless composition of multiple effects, while retaining (unlike monads) applicative style of programs. Coupled with handlers [9] which give programmers ability to interpret effects, they provide a great tool for abstracting over a set of operations which some program may perform and separating this interface from semantics of those operations defined as effect handlers.

As these features are highly desirable many calculi and languages have been developed in order to get them just right; most notable of them being: *Koka*[7] (featuring type inference, effect polymorphism with row-types and *Javascript*-like syntax), *Links*[4] (featuring *ML*-like syntax, row-typed effect polymorphism and ad-hoc effects) and *Eff*[1] with implicit effect checking and recent work on direct compilation to *OCaml*[5]. On more theoretical side, various approaches to semantics of algebraic effects can be spotted in literature, both in respect to type system and run-time semantics. Although most calculi use some form of row-types to implement tracking of effects there are differences in permitted shapes (at most one effect of given type or many effects), whether effects must be defined before use or not and how effects interact with polymorphism and abstraction. At run-time handlers can wrap the captured continuation (giving so-called deep handlers) or not (shallow handlers) and the very act of finding the right handler can be implemented in various ways, mainly depending on some constructs which skip handlers.

All this variety naturally invites Us to experiment with different features and components of a calculus. In this thesis I will build such a calculus, describing and justifying my choices and discussing the trade-offs I faced. In order to rapidly iterate on design and test the calculus, I decided to use `PLT Redex` library which allows for building language model with executable type system judgments and reduction relation. As such the other goal of my thesis was to assess viability of `PLT Redex` for development of bigger calculi. To briefly summarize my development, the calculus consists of:

- Curry style type system with ad-hoc effects in the style of *Links*, effect rows based on *Koka* and *lift* construct first shown in [2], implemented as unification based type inference algorithm.
- Executable reduction semantics most similar to system of [2].
- CEK style abstract machine with stack and *meta*-stack of handlers
- Language front-end which translates human-friendly programs to calculus terms, integrated with `Racket` environment, which allows for easy experimentation.

The rest of this thesis is structured as follows: in chapter 2 I describe the calculus in greater detail, in chapter 3 I discuss technicalities of implementation, in chapter 4 I summarize the process of integration with `Racket` environment and chapter 5 is user's manual. In the reminder of this chapter I introduce main topics of this thesis.

1.1 Algebraic effects and handlers

Algebraic effects and handlers are a language level framework which allow for coherent presentation, abstraction, composition and reasoning about computational effects. The key idea is to separate invocation of an effectful operation in an expression from the meaning of such operation. When one invokes an operation, current continuation (up to nearest handler) is captured and passed along with operation's argument to nearest handler. The handler in turn may execute arbitrary expression, using the continuation once, twice, returning a function which calls the continuation or simply ignoring it. This way many control structures can be modeled and generalized by algebraic effects and appropriate handlers. For example, the exceptions can be modeled using a single operation `Throw` and a handler which either returns the result when computation succeeded or returns default value, ignoring passed continuation.

```
handle e with
| Throw () r -> // return default value
| return x   -> x
end
```

From the language design standpoint algebraic effects provide single implementation of various phenomena which may happen during execution of a program, for example mutable state, I/O, environment lookup, exceptions etc. in a sense that every effect is treated the same, the typing rules are defined for invocation of any operation, and handling of any operation. Similarly the operational semantics is also quite simple and succinct thanks to uniform treatment of various effects. This framework is also extendable. With small additions if can handle built-in effects in addition to user-defined ones.

From the language user perspective algebraic effects provide means of abstraction over effects used in a program. Thanks to easy creation of new effects, one can define special purpose operations and their handlers to better represent domain specific problems while simultaneously using well known effects, defined in standard library. With effects being tracked by the type system, programmers can enforce purity or specific set of used effects at compile-time, or using effect polymorphism they can write reusable functions which abstract over effects which may happen. The separation of definition and implementation of effects allows for various interpretations of operations, similar to a technique of *dependency-injection* used for example during testing.

1.2 Type inference

Type inference is a technique of algorithmic reconstruction of types for various constructions used in a language. It allows programmers to write programs with no type annotations, which often feel redundant and obfuscate the meaning of a program. The most well known type system with inference is a system for *ML* family of languages - *Haskell*, *OCaml*, *SML* which infers the types with no annotations whatsoever. Formal type system defines grammar of types consisting of base types (`int`, `bool` etc.), type constructors (arrows, algebraic data types) and type variables. The typing rules require types which should be compatible (f.e. formal parameter and argument types) to unify. The key feature of this system is so called let-polymorphism - generalization of types of let-bound variables. This way code reuse can be accomplished without complicating the type system and compromising type safety. The basis of implementation of this system is first order unification algorithm, which syntactically decomposes types and builds a substitution from type variables to types.

1.3 Reduction semantics and abstract machines

Reduction semantics is a format for specifying dynamic semantics of a calculus in an operational style. The basic idea is to first define redexes - expressions which can be reduced, and contexts in which the reduction can happen. Taking λ-calculus with call-by-value reduction order as an example, the only redex is application of a function to value $(\lambda x.e)v$. The possible contexts are: empty context $\square$ or evaluation of operator part of application Ee or evaluation of operand vE. With these possibilities in mind, we will define binary relation $\longrightarrow$ which describes single step of reduction. Such relation can be thought of as a transition system, rewriting terms into simpler ones step by step. There usually are two approaches to definition of such relation:

v ::= (λ x e) | *number*
e ::= v | (e e) | x
x ::= *variable-not-otherwise-mentioned*
E ::= [] | (E e) | (v E)

Figure 1.1: λ-calculus abstract syntax

E[((λ x e) v)] $\longrightarrow$ E[substitute⟦e, x, v⟧] [β]

Figure 1.2: λ-calculus reduction relation

- Definition of primitive reduction $(\lambda x.e)v \longrightarrow_p e\{v/x\}$ which operates only on redexes and giving it a closure via following inference rule: $\dfrac{e \longrightarrow_p e'}{E[e] \longrightarrow E[e']}$, which says that if we can primitively reduce some expression, than we can do it in any context.
- Or definition of $\longrightarrow$ directly, with decomposition of terms on both sides: $E[(\lambda x.e)v] \longrightarrow E[e\{v/x\}]$

where the syntax $e\{v/x\}$ means term e with value v substituted for variable x, and $E[e]$ means some context E with expression e inserted into the hole. For both approaches it is important, that any term can be uniquely decomposed into redex and context, because when it is the case, then the relation is deterministic and gives good basis for formulation of abstract machines, interpreters or transformations to some other intermediate representations.

Abstract machine is a mathematical construction, usually defined as a set of configurations with deterministic transformations, which are computationally simple. The goal for formulation of an abstract machine is to mechanize evaluation of terms while retaining semantics given in more abstract format, f.e. reduction semantics, with the correspondence being provable[3]. As an example I will show a *CEK*-machine for the λ-calculus defined earlier. The name *CEK* comes from *C*ommand, *E*nvironment and *K*ontinuation. The machine configuration is a triple (e, ρ, κ) where e is an expression which is decomposed or reduced, ρ is an environment mapping variables to values, and the last component κ is a continuation stack, which determines what will happen with value, to which first component eventually reduces. Thanks to the environment we no longer have to explicitly perform substitution, leading to more machine friendly and efficient implementation. Given an initial state, the machine can then repeatedly apply transformation relation, either looping, arriving at a final value, or getting stuck.

1.4 PLT Redex

λ-calculus reduction example

Figure 1.3: λ-calculus example reduction sequence

CEK-machine for λ-calculus

Figure 1.4: λ-calculus abstract machine

Chapter 2

The calculus

The calculus implemented in this thesis is based on lambda calculus with call-by-value semantics. This choice follows other calculi which allow for computational effects, because fixed evaluation order is essential to obtaining sane program semantics. Inspired by *Links* [4] the operations are truly ad-hoc meaning that they don't have to be declared before usage. Moreover the calculus requires no type annotations whatsoever in spirit of *ML* family of languages while still tracking effects which occur in a program. The λ-calculus is extended with base types (Numeric and Boolean) with corresponding operations and literals, conditional expression and recursive functions. Effectful operations are invoked similarly to function calls (although they are a distinct syntactic category) and are handled using *handle* construct, they may also be lifted with *lift* syntactic form. Usually the operation will be handled by the closest handler with appropriate case, unless there is a lift for this operation in between, each one causing operation to skip one handler. Abstract syntax of the calculus is presented in Figure 2.1

2.1 Static semantics

The type system is based on *Koka*(Leijen's style of row types[6]), *Links*[4] (ad-hoc operations) and Biernacki et al. [2] (lift construct) systems. Initially I implemented a variant of System-F extended with row-types but it proved to be a bit of a mouthful to write even simplest programs. Moreover the `Redex`'s facilities for automatic testing were not able to generate well typed terms, so some type inference was inevitable. To limit the amount of work I decided to present the calculus in Curry style, with typing relation inferring the type for unannotated terms. Building on well known foundations, types are inferred via first order unification. The system does not feature polymorphism in first class fashion, as there is no rule where types are generalized, but I believe it to be a straightforward addition, following the *Koka*[7] calculus. Still, after inferring type of an expression, we can see which unification variables are left abstract and could be generalized. There are two main features

$$
\begin{array}{rcl}
v & ::= & \mathit{number} \mid (\lambda\ x\ e) \mid (\mathsf{rec}\ x\ x\ e) \mid \mathsf{true} \mid \mathsf{false} \\
\mathit{prim} & ::= & + \mid - \mid * \mid == \mid <= \mid >= \\
e & ::= & v \mid x \mid (e\ e) \mid (\mathsf{if}\ e\ e\ e) \mid (\mathsf{fix}\ e) \\
 & & \mid (\mathit{op}\ e) \mid (\mathsf{handle}\ e\ \mathit{hs}\ \mathit{ret}) \mid (\mathsf{lift}\ \mathit{op}\ e) \mid (\mathit{prim}\ e\ \ldots) \\
\mathit{hs} & ::= & ((\mathit{op}_{!\_1}\ \mathit{hexpr})\ \ldots) \\
\mathit{hexpr} & ::= & (x_{!\_1}\ x_{!\_1}\ e) \\
\mathit{ret} & ::= & (\mathsf{return}\ x\ e) \\
h & ::= & (\mathit{op}\ \mathit{hexpr}) \\
t & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid (t\ \text{->}\ \mathit{row}\ t) \mid (t\ \text{=>}\ t) \mid \mathit{row} \mid a \\
\mathit{row} & ::= & (\mathit{op}\ t\ \mathit{row}) \mid a \mid \cdot \\
x & ::= & (\text{variable-prefix v:}) \\
a & ::= & (\text{variable-prefix t:}) \\
\mathit{op} & ::= & (\text{variable-prefix op:}) \\
\Gamma & ::= & (x\ t\ \Gamma) \mid \cdot \\
E & ::= & [] \mid (E\ e) \mid (v\ E) \mid (\mathit{prim}\ E\ e) \mid (\mathit{prim}\ v\ E) \mid (\mathsf{if}\ E\ e\ e) \\
 & & \mid (\mathit{op}\ E) \mid (\mathsf{handle}\ E\ \mathit{hs}\ \mathit{ret}) \mid (\mathsf{lift}\ \mathit{op}\ E) \\
S & ::= & (a\ t\ S) \mid \cdot \\
N, n & ::= & \mathit{natural} \\
\mathit{SN} & ::= & (S\ N)
\end{array}
$$

Figure 2.1: Abstract syntax

differentiating this system from *Koka*'s; firstly effects need not be defined before use, their signature is inferred as with any other construction; secondly the system is algorithmic, with rules explicitly encoding a recursive function which can infer the type of an expression.

2.1.1 Row-types

Row types as described in [6]

2.1.2 Type inference

The judgment $(infer\, \Gamma\, [S_1\, N_1]\, e\, t\, row\, [S_2\, N_2])$ asserts that in typing context Γ under type substitution S, with name supply state N_1 expression e has type t with effects row under type substitution S_2 and with name supply state S_2. The main judgment *infer* infers a type and effect row, and calculates new substitution, given typing environment, current substitution and an expression. As in *ML* languages only simple types can be inferred, along with effect rows

2.1.3 Effect handlers

2.2 Reduction semantics

2.3 Abstract machine

$$\frac{\text{fresh-row}[\![N_1, row, N_2]\!]}{\Gamma \mid [S\ N_1] \vdash \text{true} : \text{Bool} \mathbin{!} row \mid [S\ N_2]}$$

$$\frac{\text{fresh-row}[\![N_1, row, N_2]\!]}{\Gamma \mid [S\ N_1] \vdash \text{false} : \text{Bool} \mathbin{!} row \mid [S\ N_2]}$$

$$\frac{\text{fresh-row}[\![N_1, row, N_2]\!]}{\Gamma \mid [S\ N_1] \vdash number : \text{Int} \mathbin{!} row \mid [S\ N_2]}$$

$$\frac{\text{lookup}[\![\Gamma, x, t]\!] \qquad \text{fresh-row}[\![N_1, row, N_2]\!]}{\Gamma \mid [S\ N_1] \vdash x : t \mathbin{!} row \mid [S\ N_2]}$$

$$\frac{\begin{array}{c}\text{fresh-var}[\![N_1, t_1, N_2]\!] \qquad \text{fresh-row}[\![N_2, row_2, N_3]\!] \\ (x\ t_1\ \Gamma) \mid [S_1\ N_3] \vdash e : t_2 \mathbin{!} row_1 \mid SN\end{array}}{\Gamma \mid [S_1\ N_1] \vdash (\lambda\ x\ e) : (t_1 \text{ -> } row_1\ t_2) \mathbin{!} row_2 \mid SN}$$

$$\frac{\begin{array}{c}\text{fresh-arr}[\![N_1, t_1, \text{->}, row_1, t_2, N_2]\!] \qquad \text{fresh-row}[\![N_2, row_2, N_3]\!] \\ (x_f\ (t_1 \text{ -> } row_1\ t_2)\ (x_a\ t_1\ \Gamma)) \mid [S_1\ N_3] \vdash e : t \mathbin{!} row \mid SN_1 \\ SN_1\ row_1 \sim row\ SN_2 \qquad SN_2\ t_2 \sim t\ SN_3\end{array}}{\Gamma \mid [S_1\ N_1] \vdash (\text{rec}\ x_f\ x_a\ e) : (t_1 \text{ -> } row_1\ t_2) \mathbin{!} row_2 \mid SN_3}$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e_1 : t_a \mathbin{!} row_a \mid SN_2 \qquad \text{unify-arr}[\![SN_2, t_a, t_1, \text{->}, row_1, t_2, SN_3]\!] \\ \Gamma \mid SN_3 \vdash e_2 : t_3 \mathbin{!} row_2 \mid SN_4 \\ SN_4\ t_1 \sim t_3\ SN_5 \qquad SN_5\ row_1 \sim row_2\ SN_6 \qquad SN_6\ row_1 \sim row_a\ SN_7\end{array}}{\Gamma \mid SN_1 \vdash (e_1\ e_2) : t_2 \mathbin{!} row_2 \mid SN_7}$$

$$\frac{\text{check-prim}[\![\Gamma, SN_1, prim, (e\ \ldots), t, row, SN_2]\!]}{\Gamma \mid SN_1 \vdash (prim\ e\ \ldots) : t \mathbin{!} row \mid SN_2}$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e_{cond} : t_{cond} \mathbin{!} row_{cond} \mid SN_2 \qquad SN_2\ t_{cond} \sim \text{Bool}\ SN_3 \\ \Gamma \mid SN_3 \vdash e_{then} : t_{then} \mathbin{!} row_{then} \mid SN_4 \qquad \Gamma \mid SN_4 \vdash e_{else} : t_{else} \mathbin{!} row_{else} \mid SN_5 \\ SN_5\ t_{then} \sim t_{else}\ SN_6 \qquad SN_6\ row_{cond} \sim row_{then}\ SN_7 \qquad SN_7\ row_{then} \sim row_{else}\ SN_8\end{array}}{\Gamma \mid SN_1 \vdash (\text{if}\ e_{cond}\ e_{then}\ e_{else}) : t_{then} \mathbin{!} row_{then} \mid SN_8}$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e : t_1 \mathbin{!} row_1 \mid [S_1\ N_1] \qquad \text{fresh-row}[\![N_1, row_2, N_2]\!] \qquad \text{fresh-var}[\![N_2, t_2, N_3]\!] \\ [S_1\ N_3]\ (op\ (t_1 \Rightarrow t_2)\ row_2) \sim row_1\ SN_2\end{array}}{\Gamma \mid SN_1 \vdash (op\ e) : t_2 \mathbin{!} row_1 \mid SN_2}$$

$$\frac{\Gamma \mid SN_1 \vdash e : t \mathbin{!} row \mid [S_1\ N_1] \qquad \text{fresh-var}[\![N_1, a, N_2]\!]}{\Gamma \mid SN_1 \vdash (\text{lift}\ op\ e) : t \mathbin{!} (op\ a\ row) \mid [S_1\ N_2]}$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e : t_1 \mathbin{!} row_1 \mid SN_2 \qquad (x\ t_1\ \Gamma) \mid SN_2 \vdash e_{ret} : t_{ret} \mathbin{!} row_{ret} \mid SN_3 \\ \text{infer-handlers}[\![\Gamma, SN_3, t_{ret}, hs, row_{out}, row_{handled}, SN_4]\!] \qquad SN_4\ row_{out} \sim row_{ret}\ SN_5 \\ SN_5\ row_1 \sim row_{handled}\ SN_6\end{array}}{\Gamma \mid SN_1 \vdash (\text{handle}\ e\ hs\ (\text{return}\ x\ e_{ret})) : t_{ret} \mathbin{!} row_{out} \mid SN_6}$$

Figure 2.2: Type system

$$
\begin{array}{rcll}
E[((\lambda\ x\ e)\ v)] & \longrightarrow & E[\text{substitute}[\![e,\ x,\ v]\!]] & [\beta\text{-}\lambda] \\
E[((\text{rec}\ x_f\ x_a\ e)\ v)] & \longrightarrow & E[\text{substitute}[\![\text{substitute}[\![e,\ x_f,\ (\text{rec}\ x_f\ x_a\ e)]\!],\ x_a,\ v]\!]] & [\beta\text{-rec}] \\
E[(prim\ v_1\ v_2)] & \longrightarrow & E[\text{prim-apply}[\![prim,\ v_1,\ v_2]\!]] & [\text{prim-op}] \\
E[(\text{if true}\ e_1\ e_2)] & \longrightarrow & E[e_1] & [\text{if-true}] \\
E[(\text{if false}\ e_1\ e_2)] & \longrightarrow & E[e_2] & [\text{if-false}] \\
E[(\text{lift}\ op\ v)] & \longrightarrow & E[v] & [\text{lift-compat}] \\
E[(\text{handle}\ v\ hs\ (\text{return}\ x\ e))] & \longrightarrow & E[\text{substitute}[\![e,\ x,\ v]\!]] & [\text{handle-return}] \\
E_1[(\text{handle}\ E_2[(op\ v)]\ hs\ ret)] & \longrightarrow & E_1[\text{substitute}[\![\text{substitute}[\![e,\ x_1,\ v]\!], & [\text{handle-op}] \\
 & & \quad x_2,\ (\lambda\ \text{v:z}\ (\text{handle}\ E_2[\text{v:z}]\ hs\ ret))]\!]] & \\
 & & \multicolumn{2}{l}{\text{where free}[\![op,\ E_{in},\ 0]\!],\ \text{get-handler}[\![op,\ hs,\ (x_1\ x_2\ e)]\!],\ \text{v:z fresh}}
\end{array}
$$

Figure 2.3: Reduction relation

Chapter 3

Implementation

3.1 `PLT Redex`

3.2 Typing relation

3.3 Unification

3.4 Reduction relation

3.5 Automatic testing

Chapter 4

The `Racket` environment

4.1 Front-end

4.2 Back-end

Chapter 5

User's manual

Bibliography

[1] Andrej Bauer and Matija Pretnar. ?Programming with Algebraic Effects and Handlers? In: *CoRR* abs/1203.1539 (2012). arXiv: `1203.1539`. URL: `http://arxiv.org/abs/1203.1539`.

[2] Dariusz Biernacki et al. ?Handle with Care: Relational Interpretation of Algebraic Effects and Handlers? In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 8:1–8:30. ISSN: 2475-1421. DOI: `10.1145/3158096`. URL: `http://doi.acm.org/10.1145/3158096`.

[3] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.

[4] Daniel Hillerström and Sam Lindley. ?Liberating Effects with Rows and Handlers? In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: ACM, 2016, pp. 15–27. ISBN: 978-1-4503-4435-7. DOI: `10.1145/2976022.2976033`. URL: `http://doi.acm.org/10.1145/2976022.2976033`.

[5] Oleg Kiselyov and KC Sivaramakrishnan. ?Eff Directly in OCaml? In: *arXiv e-prints*, arXiv:1812.11664 (Dec. 2018), arXiv:1812.11664. arXiv: `1812.11664 [cs.PL]`.

[6] Daan Leijen. ?Extensible records with scoped labels? In: *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05), Tallin, Estonia*. Sept. 2005. URL: `https://www.microsoft.com/en-us/research/publication/extensible-records-with-scoped-labels/`.

[7] Daan Leijen. ?Koka: Programming with Row Polymorphic Effect Types? In: *Mathematically Structured Functional Programming 2014*. EPTCS, Mar. 2014. URL: `https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/`.

[8] Gordon Plotkin and John Power. ?Algebraic Operations and Generic Effects? In: *Applied Categorical Structures* 11.1 (Feb. 2003), pp. 69–94. ISSN: 1572-9095. DOI: `10.1023/A:1023064908962`. URL: `https://doi.org/10.1023/A:1023064908962`.

[9] Gordon Plotkin and Matija Pretnar. ?Handling Algebraic Effects? In: *Logical Methods in Computer Science* 9.4 (Dec. 2013). Ed. by Andrzej Tarlecki. ISSN: 1860-5974. DOI: `10.2168/lmcs-9(4:23)2013`. URL: `http://dx.doi.org/10.2168/LMCS-9(4:23)2013`.