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

Abstract

English abstract

Abstrakt w języku polskim

Contents

Chapter 1

Introduction

- Algebraic effects and handlers

    - What are algebraic effects
    - Why do we need handlers
    -

- Reduction semantics

- Type inference

- `PLT Redex`

    - Defining languages, relations on terms, reduction semantics
    - Automatic testing, counterexample search

Algebraic effects [1] 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 [2] 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. While designing a calculus representing a core of programming language, one must be very careful deciding which features to add and keep in mind all the interactions between them. Following most of the functional language designers I chose λ-calculus as a basis extended with operation invocation, effect lifting and handling. The type system for this calculus is defined in Curry style, which means that expressions do not contain types, which are assigned to them by the typing relation. In more practical terms I had to implement type inference, instead of type checking. Also besides the type system which specifies static semantics of calculus, one must specify the behavior of programs at run-time. I chose the format of reduction semantics, as it is quite readable and is supported by `PLT Redex` library.

1.1   Algebraic effects and handlers

Algebraic effects and handlers are a language level framework which allow for co-
herent presentation, abstraction, composition and reasoning about computational
effects. The key idea is to separate invocation of an effectful operation in an expres-
sion from the meaning of such operation. When one invokes an operation, current
continuation is captured and passed along with operation's argument to nearest han-
dler. 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 us-
ing a single operation `throw` and a handler which either returns the result when
computation succeeded or returns default value, ignoring passed continuation.

```
handler
| 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 abstrac-
tion 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 pu-
rity 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 interpreta-
tions 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 con-
structions 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.

$$e ::= x \mid \lambda x.e \mid ee$$
$$E ::= \Box \mid Ee$$

Then we have to define reduction relation, which generates a transition system for terms in the calculus. The rules usually take the form of redex in a context which reduces to some expression in a context.

Chapter 2

The calculus

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

4.0.1   Front-end

4.0.2   Back-end

Chapter 5

User's manual

Bibliography

[1] 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.

[2] 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.