Abstract

To date, there has been limited work on the semantics of languages with polymorphic effect systems. The application, by Moggi, of strong monads to modelling the semantics of effects has become a mainstream concept in functional programming languages. This was improved upon by Lucasson (?) using a graded monad to model languages with a range of independent and dependent effects at an operational level. A categorical semantics for parametric polymorphism in types was first published by Reynolds (?) allowing a denotational analysis of languages including type parameters. There has been some work on polymorphism over the exception effect (which paper). Despite these works, there has been no work to date on the denotational semantics of languages with general parametric polymorphism over effects.

In this dissertation, I present several pieces of work. Firstly, I shall introduce a modern definition of a lambda-calculus based language with an explicit graded monad to handle a variety of effects. This calculus shall then be extended using polymorphic terms to yield a more general polymorphic-effect-calculus. I shall then give an indexed-category-based denotational semantics for the language, along with an outline of a proof for the soundness of these semantics. The full proofs can be found online on my github repository (link), since due to the number of theorems and cases, the total size is well over 100 pages of definitions, theorems, and proofs.

Chapter 1

Introduction

1.1 What is Effect Polymorphism?

Effect polymorphism is when the same function in a language can operate on values of similar types but with different effects. It allows the same piece of code to be used in multiple contexts with different type signatures. This manifests in a similar manner to type parameter polymorphism in system-F based languages. Consider the following Scala-style pseudo-code:

```
def check[E: Effect](
    action: Unit => (Unit;e)
): Unit; (IO, e) {
    val ok: Boolean = promptBool(
        "Are you sure you want to do this?"
    )
    If (ok) {
        action()
    } else {
       abort()
    }
}
```

```
check[IO, RealWorld](() => check[RealWorld](FireMissiles))
check[Transaction](SendMoney(Bob, 100, USD))
check[Exception](ThrowException(\Not Aborted"))
```

In this case, we are reusing the same "check" function in three different cases with three different

effects in a type safe manner. "Check" is polymorphic in the effect parameter it receives. To analyse this language, it would be useful to have precision to compare different effects.

1.2 An Introduction to Categorical Semantics

When we specify a denotational semantics of a language in category theory, we look to find a mapping of types and typing environments to objects in a given category.

$$A : \texttt{Type} \mapsto [\![A]\!]_M \in \texttt{obj}\ \mathbb{C} \tag{1.1}$$
$$\Gamma \mapsto [\![\Gamma]\!]_M \in \texttt{obj}\ \mathbb{C} \tag{1.2}$$

Further more, instances of the type relation should be mapped to morphisms between the relevant objects.

$$\Gamma \vdash v\colon A \mapsto \mathbb{C}([\![\Gamma]\!]_M, [\![A]\!]_M) \tag{1.3}$$

This should occur in a sound manner. That is, for every instance of the $\beta\eta$-equivalence relation between two terms, the denotations of the terms should be equal in the category.

$$\Gamma \vdash v_1 =_{\beta\eta} v_2\colon A \implies [\![\Gamma \vdash v_1\colon A]\!]_M = [\![\Gamma \vdash v_2\colon A]\!]_M \tag{1.4}$$

To prove soundness, we induct over derivation of the $\beta\eta$-equivalence relation. In doing so, we need to prove a couple of properties which help prove cases. These are substitution and weakening. **TODO: What do these do?**

1.2.1 Arrows and Objects

1.2.2 Languages and Their Requirements

One of the simplest, while still interesting, languages to derive a denotational semantics for is the simply typed lambda calculus (STLC). STLC's semantics require a cartesian closed category (see section 2.1).

Products are used to denote the lists of variable types in the typing environment, exponential objects model functions, and the terminal object is used to derive "points" (**TODO: Terminology**) which represent the ground terms, such as the unit term, (), as well as the empty typing environment.

From this, we can specify what structures categories need to have in order to model more complex languages.

Language Feature	Structure Required
STLC	CCC
Single Effect	Strong Monad
Multiple Effects	Strong Graded Monad
Polymorphism	Indexed Category

A single effect can be modelled using a strong monad, as shown by Moggi **TODO: Reference**. The monad allows us to generate a unit effect and to compose multiple instances of the effect together.

For more precise analysis of languages with multiple effects, we can look into the algebra on the effects. A simple algebra is a partially ordered monoid. The monoid operation defines how to compose effects, and the partial order gives a sub-typing relation to make programming more intuitive. A strong graded monad allows us to model this algebra in a category theoretic way.

To express polymorphism over a property P, the language's semantics are expanded to use a new environment specifying the variables ranging over P that are allowed in a given context. To model this, we can create a category representing the semantics of the non-polymorphic language at each given context, indexed by a base category which models the operations and relationships between the P-Environment. Morphism in the base category between environments correspond to functors between the semantic categories for the relevant environments. These functors can then be used to construct the semantics of polymorphic terms. **TODO: Index diagram**

Chapter 2

Required Category Theory

Before going further, it is assert a common level of category theory knowledge.

2.1 Cartesian Closed Category

Recall that a category is cartesian closed if it has a terminal object, products for all pairs of objects, and exponentials for all objects.

2.1.1 Terminal Object

2.1.2 Products

2.1.3 Exponentials

2.2 Co-Product

2.3 Functors

A functor $F : \mathbb{C} \to \mathbb{D}$ is a mapping of objects:

$$A \in \texttt{obj}\ \ \mathbb{C} \mapsto FA \in \texttt{obj}\ \ \mathbb{D} \tag{2.1}$$

And morphisms:

$$f : \mathbb{C}(A, B) \mapsto F(f) : \mathbb{D}(FA, FB) \tag{2.2}$$

that preserves the category properties of composition and identity.

$$F(\texttt{Id}_A) = \texttt{Id}_{FA} \tag{2.3}$$
$$F(g \circ f) = F(g) \circ F(f) \tag{2.4}$$

2.4 Monad

2.5 Graded Monad

2.6 Tensor Strength

2.7 Adjunction

2.8 Strict Indexed Category