CALL-BY-PUSH-VAUE WITH ALGEBRAIC EFFECTS AND HANDLERS

Student: Stavros Alexandros Dimakos (S.A.Dimakos@student.tudelft.nl) Supervisor: Jaro Reinders Responsible professor: Casper Bach Poulsen

1. Background

- **Algebraic Effects and Handlers:** A programming model for modularly defining and handling computational effects such as state, exceptions, or I/O in a programming language.
- **Call-by-Push-Value (CBPV):** A programming language evaluation strategy that decomposes both call-by-value and call-by-name paradigms into more primitive operations, providing a unified framework that can simulate both call-by-value and call-by-name behaviors [1].
- **Extended Call-by-Push-Value:** An enhancement of CBPV that includes additional constructs to include lazy evaluation [2].
- **Research Gap:** Practical implementation of (CBPV) in mainstream programming languages remains limited.

2. Research Question

How can algebraic effects and handlers be used to construct an interface capable of achieving call-by-push-value in Haskell?

Sub-questions explored in the research:

- What are the benefits of call-by-push-value?
- How can we create an interface that can be used to translate programs to different evaluation strategies?
- How can we define the behavior of the implementation using mathematical laws?
- How can the implementation be proven correct w.r.t. its laws?
- How closely does the implementation and laws align with existing theory found in the literature?

3. Free Monad

An **algebraic effect** can be described as an interface that includes a collection of associated operations. First introduced in the context of category theory, the **free monad**, Free f a, models these operations as **abstract syntax trees**, with nodes for operations (Op) and values (Pure):

```
data Free f a
        = Pure a
        | Op (f (Free f a))
```

The Free Monad provides a flexible structure to encode a variety of **operations** and enables the modeling of **side effects** [3]. Our interface leverages the Free Monad to showcase how different evaluation regimes interact with these diverse **operations** and **side effects**.

4. Thunk

The implementation of **thunk** is the first step in our interface to manage delayed computations typical of **call-by-name** and **call-by-need evaluation** strategies. This is composed of two functions:

```
thunk :: Free f a -> Free f (Thunk f a)
force :: Thunk f a -> Free f a
```

The **thunk** function encapsulates computations in a data structure until needed, and the **force** function triggers immediate computation. To accommodate various evaluation strategies, three versions of each function have been implemented: **CBName**, **CBValue**, and **CBNeed**. For call-by-need, **memoization** ensures each thunk is evaluated only once and described by the following function signature:

```
thunkCBNeed :: State [Pack] < f => Free f a -> Free f t
forceCBNeed :: State [Pack] < f => t -> Free f a
```

In the type signatures we can see the following elements:

- **t:** A thunked computation.
- **State:** A signature functor that defines a Put and a Get operation emulating memory.
- **Pack:** An integer value pair to store evaluated expressions.

5. Translation of Types and Terms

Using the implementation of **thunks**, we can now translate operations to different evaluation strategies. To demonstrate this, we define a **denote** function that maps the syntax of a lambda calculus-based language onto effectful operations. By utilizing different **thunk** and **force** functions in our **denote** function, we can alter the evaluation order. For the translation of the language, we refer to the theory in Levy's paper [1].

$A_0, \dots, A_{n-1} \vdash M : C$	$UA_0^{\mathtt{n}}, \dots, UA_{n-1}^{\mathtt{n}} \vdash^{\mathsf{c}} M^{\mathtt{n}} : C^{\mathtt{n}}$
$\mathtt{x}$	$\mathtt{force}\ \mathtt{x}$
$\mathtt{let}\ \mathtt{x}\ \mathtt{be}\ M.\ N$	$\mathtt{let}\ \mathtt{x}\ \mathtt{be}\ \mathtt{thunk}\ M^{\mathtt{n}}.\ M^{\mathtt{n}}$
$\mathtt{true}$	$\mathtt{produce}\ \mathtt{true}$
$\mathtt{false}$	$\mathtt{produce}\ \mathtt{false}$
$\mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N'$	$M^{\mathtt{n}}\ \mathtt{to}\ \mathtt{z}.\ \mathtt{if}\ \mathtt{z}\ \mathtt{then}\ N^{\mathtt{n}}\ \mathtt{else}\ N'^{\mathtt{n}}$
$\mathtt{inl}\ M$	$\mathtt{produce}\ \mathtt{inl}\ \mathtt{thunk}\ M^{\mathtt{n}}$
$\mathrm{pm}\ M\ \mathtt{as}\ \{\mathtt{inl}\ \mathtt{x}.N, \mathtt{inr}\ \mathtt{x}.N'\}$	$M^{\mathtt{n}}\ \mathtt{to}\ \mathtt{z}.\ \mathrm{pm}\ \mathtt{z}\ \mathtt{as}\ \{\mathtt{inl}\ \mathtt{x}.N^{\mathtt{n}}, \mathtt{inr}\ \mathtt{x}.N'^{\mathtt{n}}\}$
$\lambda \mathtt{x}.M$	$\lambda \mathtt{x}.M^{\mathtt{n}}$
$N`M$	$(\mathtt{thunk}\ N^{\mathtt{n}})`M^{\mathtt{n}}$
$\mathtt{print}\ c;\ M$	$\mathtt{print}\ c;\ M^{\mathtt{n}}$

Figure 1: Translation of CBN types and terms (Levy Paul Blain, 2001, p.56).

6. Laws

We use **mathematical laws** from existing research to ensure our implementation's **correctness**. These laws make the execution of programs written against our interface predictable and enable **direct reasoning** about programs across different evaluation strategies, beyond the meta-level abstraction.

(a) β/η laws

$$
\begin{aligned}
\text{thunk}\ m \gg\!= \text{force} &\equiv m \\
\text{Let}\ x\ v\ m &\equiv \text{App}\ v\ (\text{Lam}\ x\ m) \\
m &\equiv \text{Let}\ x\ m\ (\text{Var}\ x) \\
\text{Pm}\ (\text{Inl}\ v)\ (\text{Inl}\ (\text{Lam}\ x\ m_1))\ (\text{Inr}\ (\text{Lam}\ x\ m_2)) &\equiv \text{App}\ v\ (\text{Lam}\ x\ m_1) \\
\text{Pm}\ (\text{Inr}\ v)\ (\text{Inl}\ (\text{Lam}\ x\ m_1))\ (\text{Inr}\ (\text{Lam}\ x\ m_2)) &\equiv \text{App}\ v\ (\text{Lam}\ x\ m_2)
\end{aligned}
$$

(b) Sequencing laws

$$
\begin{aligned}
\text{Let}\ y\ (\text{Let}\ x\ m\ n)\ p &\equiv \text{Let}\ x\ m\ (\text{Let}\ y\ n\ p) \\
\text{Let}\ x\ m\ (\text{Lambda}\ y\ n) &\equiv \text{Lambda}\ y\ (\text{Let}\ x\ m\ n)
\end{aligned}
$$

Figure 2: Equational theory

We prove a set of **β** and **η-reduction** laws that ensure different expressions representing the same **computation** or **value** are treated **equivalently**, as well as some **sequential** laws that enable restructuring of expressions without breaking **equivalence**.

7. Limitations and Future Work

Limitations

- Creating **modular components** for operations described by algebraic effects is **not always feasible**.
- **Complex interactions** between effects often require **global knowledge**, **undermining modularity**.
- Correctly **scoping effect handlers** in a modular system is challenging.
- Potential for unintended behavior or **performance issues** due to scoping challenges.

Future Work

- Programmers can **reason** about programs written with different evaluation strategies.
- Discover **equivalences** within and between evaluation strategies based on implemented effects.
- Facilitate **optimization** and **analysis of performance** across different evaluation strategies for various programs.

Bibliography:

[1] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, Typed Lambda Calculi and Applications, pages 228–243, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg

[2] Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In Lu´ıs Caires, editor, Programming Languages and Systems, pages 235–262, Cham, 2019. Springer International Publishing.

[3] WOUTER SWIERSTRA. Data types `a la carte. Journal of Functional Programming, 18(4):423–436, 2008.