Homework (DM)
Compilation and Program Analysis (CAP)

Algebraic Effects and Effect Handlers

Instructions:

1. Every single answer must be informally explained AND formally proved.
2. Using LaTeX is NOT mandatory at all.
3. Vous avez le droit de rédiger en Français.

In this Homework, you will study algebraic effects and handlers. The language we will consider differs in nature from **WHILE**: it has a rather functional nature, as opposed to imperative. Algebraic effects are a relatively modern creation, essentially popularized by Matija Pretnar's PhD thesis in 2010. They have since then percolated into real world languages, and are notably part of OCaml 5.

1 Preliminaries: lambda-calculus

We first introduce λ, a simple purely functional programming language whose syntax is depicted on Figure 1. λ is quite standard, except perhaps that values and computations are explicitly split into two categories. Values are "inert", they do not reduce, and include $()$, the only value of the `unit` type, booleans, natural numbers, variables, and (anonymous) functions. Computations are proper programs: they include return statements, sequences, conditionals, and function application. Figure 2 defines λ's (small step) operational semantics. The last four rules describe the reduction of the computations, while the first one lifts this reduction under evaluation contexts.

Values:

$$
\begin{array}{rll}
v \in \mathcal{V} ::= & () \mid tt \mid ff \mid n & \text{Constants} \\
& \mid \mathbf{x} & \text{Variables} \\
& \mid \lambda \mathbf{x}.c & \text{Functions}
\end{array}
$$

Computations:

$$
\begin{array}{rll}
c \in \mathcal{C} ::= & \texttt{return}\ v & \text{Return} \\
& \mid \texttt{let}\ \mathbf{x} = c_1\ \texttt{in}\ c_2 & \text{Sequence} \\
& \mid \texttt{if}\ v\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} & \text{Conditional} \\
& \mid v_1\ v_2 & \text{Application}
\end{array}
$$

Figure 1: Grammar for λ

Small step relation for λ: $c \rightsquigarrow c'$
We define the evaluation contexts C as:

$$C ::= [] \mid \texttt{let}\ \mathbf{x} = C\ \texttt{in}\ c$$

$$\frac{c \rightsquigarrow c'}{C[c] \rightsquigarrow C[c']} \qquad \frac{}{\texttt{let}\ \mathbf{x} = \texttt{return}\ v\ \texttt{in}\ c_2 \rightsquigarrow c_2[v/\mathbf{x}]} \qquad \frac{}{\texttt{if}\ tt\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \rightsquigarrow c_1}$$

$$\frac{}{\texttt{if}\ ff\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \rightsquigarrow c_2} \qquad \frac{}{(\lambda \mathbf{x}.c)\ v \rightsquigarrow c[v/\mathbf{x}]}$$

Figure 2: Operational semantics for λ

2 A calculus with effects: λ_{eff}

The core idea of algebraic effects is to think of your programs as purely functional programs, but with the ability to ask questions to the environment through operations: impure behaviour arises from a set of operations.

For instance, suppose your program interact with a memory cell containing a single bit of information. You would traditionally think of your program as a function taking a boolean as input, your state, and returning a new boolean, the updated state. Rather, we will think of our programs as a series of interaction with this cell through an interface of two operations: `get` to read the value of the cell, and `put` to update the value of the cell.

Consider now a program `flip` negating the content of the cell. It should be thought of as a sequence of two actions: `get` the content of the cell, then `put` the negation of the value read, and terminate, returning the unit value $()$. This program can be represented as something very close to an AST, as depicted on the left hand-side of Figure 3.

You may notice that this tree explains how the computation proceeds no matter which value is read, but does not explain how to know what value should be read in a concrete run, or even what is a cell for that matter. Worst, if for some reason our `flip` program were to read the cell twice in a row before updating the value, we would result in the right hand-side of Figure 3. Here, some leaves feel weird, suggesting we could read first true and then false from our cell without any update in between.

Indeed, we have described a program that interacts with an environment through an interface of operations, but we have not described the environment! This is where effect handlers come into

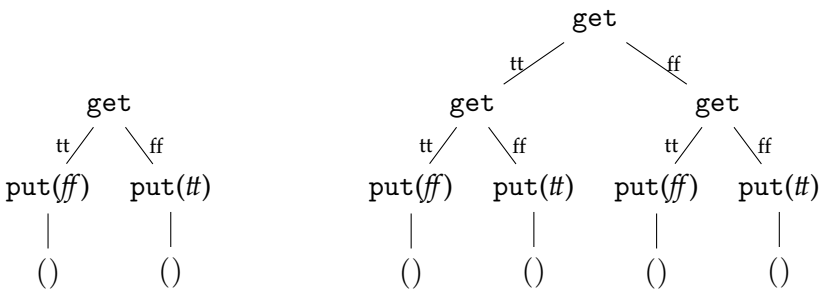

Figure 3: Flipping the content of a cell: tree representation

Values:

$$v \in \mathcal{V} ::= \ldots \mid h \quad \text{handler}$$

Computations:

$$c \in \mathcal{C} ::= \ldots \mid \texttt{with}\ h\ \texttt{handle}\ c \quad \text{Handling}$$

Handlers:

$$h \in \mathcal{H} ::= \texttt{handler}\ \{\ \texttt{return}\ \mathbf{x} \mapsto c_r, \quad \text{Return clause}$$
$$\texttt{op}_1(\mathbf{x}, \mathbf{k}) \mapsto c_1; \ldots; \texttt{op}_n(\mathbf{x}, \mathbf{k}) \mapsto c_n\} \quad \text{Operational clauses}$$

Figure 4: Grammar for $\lambda_{\textbf{eff}}$

Small step relation for $\lambda_{\textbf{eff}}$: $c \rightsquigarrow c'$
In the following rules, we set $h = \texttt{handler}\ \{\texttt{return}\ \mathbf{x} \mapsto c_r, \texttt{op}_1(\mathbf{x}, \mathbf{k}) \mapsto c_1; \ldots; \texttt{op}_n(\mathbf{x}, \mathbf{k}) \mapsto c_n\}$

$$\frac{}{\texttt{with}\ h\ \texttt{handle}\ \texttt{return}\ v \rightsquigarrow c_r[v/\mathbf{x}]}$$

$$\frac{}{\texttt{with}\ h\ \texttt{handle}\ C[\texttt{perform}\ \texttt{op}_i(v)] \rightsquigarrow c_i[v/\mathbf{x}, \lambda\mathbf{y}.\texttt{with}\ h\ \texttt{handle}\ C[\texttt{return}\ \mathbf{y}]/\mathbf{k}]}$$

Figure 5: Operational semantics for $\lambda_{\textbf{eff}}$

play: they provide implementations to the operations. They can be thought of as an extension of exception handling: the control flow is passed to a harness catching the operations, and deciding what to do in its lieu. For instance, the handler may implement our get and put effects as mentioned above, by explicitly passing as argument the cell.

A syntax for $\lambda_{\textbf{eff}}$. Let's now package these ideas into a calculus. Figure 4 extends λ's syntax with two new kind of computations. One can perform an operation op over an argument v with the computation `perform` $\texttt{op}(v)$. Furthermore, computations can be wrapped into an effect handler. This handler specifies how the computation must proceed when one of two situations arises. If the computation returns, the `Return` clause triggers. When an operation is performed, the corresponding `Operational clause` happens. In this case, the corresponding computation c_i depends on the argument of the operation (bound to $\mathbf{x}$), but also on the continuation of the

computation (bound to **k**): contrary to exception handlers, we usually want to get back to our main computation after falling in our harness! Figure 5 explicit these new semantic rules.

Conventions: to simplify the writing of programs, we introduce the following notations:

- $\lambda_{\textbf{eff}}$ variables are in bold: $\mathbf{x}$.
- When the `return` close of a handler is omitted, it is assumed to be the identity, i.e., $\lambda\mathbf{x}.\texttt{return}\ \mathbf{x}$.
- We write op instead of $op(())$ when an operation expects no argument, such as `get`.
- We write $c_1;;c_2$ for $\texttt{let}\ \mathbf{x} = c_1\ \texttt{in}\ c_2$ if $\mathbf{x}$ does not appear in c_2.
- We will allow ourselves to use data-types such as pairs or lists in the remaining even if they have not been formally defined in the language.

We can finally rewrite `flip` in $\lambda_{\textbf{eff}}$:

$$\texttt{flip} \triangleq \texttt{let}\ \mathbf{x} = \texttt{perform}\ \texttt{get}()\ \texttt{in}\ \texttt{perform}\ \texttt{put}(\mathbf{x})$$

. If we tried to evaluate `flip` according to our operational semantics, we would be immediately stuck. We hence implement a handler that passes around the boolean cell.

$$h_{cell} \triangleq \texttt{handler}\ \left\{\begin{array}{c}\texttt{return}\ \mathbf{x} \mapsto \lambda\mathbf{b}.\texttt{return}\ \mathbf{x} \\ \texttt{get}(\mathbf{k}) \mapsto \lambda\mathbf{b}.(\mathbf{k}\ \mathbf{b})\ \mathbf{b} \\ \texttt{put}(\mathbf{x},\mathbf{k}) \mapsto \lambda\mathbf{b}.(\mathbf{k}\ ())\ \mathbf{x}\end{array}\right\}$$

In the implementation of `get`, the current value of the cell is passed to the continuation, and the value of the cell is left unchanged. For `put`, we simply pass $()$, but update the value of the cell. We can finally get a full computation, which expects an initial cell to run c_i:

$$(\texttt{with}\ h_{cell}\ \texttt{handle}\ \texttt{flip})\ c_i$$

Question #1

Write a $\lambda_{\textbf{eff}}$ computation c_{and} that takes a natural number n as argument, and returns whether n is odd if the cell is set to true, and whether n is even if the cell is set to false.

Question #2

Assuming $\lambda_{\textbf{eff}}$ is extended with lists, propose an alternate handler for `get` and `put` denoted h_{log} that logs the history of all booleans that are read via `get` and returns this history at the end of the computation.

Question #3

Consider

$$c_{ex} \triangleq \texttt{let}\ \mathbf{x} = \texttt{get}\ \texttt{in}\ \texttt{let}\ \mathbf{y} = \texttt{get}\ \texttt{in}\ \texttt{put}\ \mathbf{x};;\ \texttt{return}\ \texttt{orb}\ \mathbf{x}\ \mathbf{y}$$

where `orb` is the boolean disjunction. Show the traces of execution of $\texttt{with}\ h_{cell}\ \texttt{handle}\ c_{ex}$ and $\texttt{with}\ h_{log}\ \texttt{handle}\ c_{ex}$ applied to an initial cell set to true.

Question #4

Consider now a new effect: failure. To do so, we introduce a new operation `fail` whose handler discards the continuation:

$$h_{fail} \triangleq \texttt{handler } \{\texttt{fail}(\mathbf{k}) \mapsto \texttt{return } ()\}$$

We can now consider two ways to nest handlers: `with` h_{fail} `handle with` h_{log} `handle` c or conversely `with` h_{log} `handle with` h_{fail} `handle` c.
Are those equivalent? Justify your answer either by an informal argument, or a counter-example.

Question #5

We now consider the case of non-determinism through an operation `toss`() which picks randomly a boolean. Propose an interface and handler h_{toss} implementing the non-determinism by collecting all possible results.

2.1 Algebraic Simplification

We have informally qualified some handlers as correct or incorrect. This intuition has a formal ground, hinted at the name of algebraic effects. Operations form an algebra, i.e., their signature comes with a set of equations that axiomatize their behavior.

For instance, let us consider the case of `get`/`put`. We may axiomatize the interaction between each pair of operations, i.e.:

$$\texttt{put } v_1;;\texttt{put } v_2;;c \equiv \texttt{put } v_2;; \tag{1}$$

$$\texttt{put } v;;\texttt{let } \mathbf{x} = \texttt{get in } c \equiv \texttt{put } v;;c[v/\mathbf{x}] \tag{2}$$

$$\texttt{let } \mathbf{x} = \texttt{get in put } \mathbf{x};c \equiv \texttt{let } \mathbf{x} = \texttt{get in } c \tag{3}$$

$$\texttt{let } \mathbf{x} = \texttt{get in let } \mathbf{y} = \texttt{get in } c \equiv \texttt{let } \mathbf{x} = \texttt{get in } c[\mathbf{x}/\mathbf{y}] \tag{4}$$

Spelled out, these equations state that (1) two consecutive puts reduce to the latest one, (2) a put determines uniquely the result of a get, (3) putting in memory the same cell as the one we have read is useless, and (4) two consecutive gets can be condensed to a single one.

Some care must be taken in this definition: when introducing a piece of state such as with h_{cell}, the equality $c_1 \equiv c_2$ should be understood as $(\texttt{with } h_{cell} \texttt{ handle } c_1)\ s_i$ and $(\texttt{with } h_{cell} \texttt{ handle } c_2)\ s_i$ reduce to the same value for any initial state s_i.

"Correct" handlers for `get`/`put` are hence defined in the rest of this section as the ones that satisfy these four equations.

Question #6

Propose an alternate handler h_{wrong} for `get` and `put` that would be blatantly semantically incorrect.

Question #7

Check that h_{cell} is correct. Is h_{log} correct?

Question #8

Prove that this axiomatization is not minimal by deriving (4) from the three other equations.

Question #9

These equations can be an opportunity for optimization! Write a program transformation that exploits the cell algebra. This transformation should be defined by mutual induction on the syntax of values and computations, i.e.:

$$\begin{aligned} [\![()]\!]_v &= () && \dots \\ [\![v_1\ v_2]\!]_c &= [\![v_1]\!]_v\ [\![v_2]\!]_v && \dots \end{aligned}$$

Justify its correctness informally—in particular, which hypothesis must you impose on the handlers used in your program?

Question #10 (Difficult)

Open question

Does your transformation optimize the following program:

$$\texttt{let}\ \mathbf{x} = \texttt{get}\ \texttt{in}\ \texttt{let}\ \mathbf{y} = 2 * \mathbf{x}\ \texttt{in}\ \texttt{let}\ \mathbf{z} = \texttt{get}\ \texttt{in}\ \texttt{return}\ \mathbf{y} + \mathbf{z}.$$

Discuss how you could improve the transformation to better optimize programs.

Question #11

Recall the non-determinism operation from Question #4: extend it with equations to suggest an algebra of non-determinism.

3 Compilation to CPS

We now wish to compile our language to another without support for algebraic effects; for instance Javascript, to run our programs in the browser! For this purpose, we must translate algebraic effects into simpler construct. Here, we will target simple functions, using the so-called Continuation Passing Style (CPS). In continuation passing style, the control flow is made explicit via an argument, usually denoted **k** and called the **k**ontinuation, which is the future of the execution. For instance, let us consider the program $\texttt{flip} \triangleq \texttt{let}\ \mathbf{x} = \texttt{perform}\ \texttt{get}()\ \texttt{in}\ \texttt{perform}\ \texttt{put}(\mathbf{x})$ defined earlier.

$$\texttt{flip}_{cps} \triangleq \lambda\mathbf{k}.\lambda\mathbf{k}_{\texttt{get}}.\lambda\mathbf{k}_{\texttt{put}}.(\mathbf{k}_{\texttt{get}}\ (\lambda\mathbf{x}.\mathbf{k}_{\texttt{put}}\ (\mathbf{x},\mathbf{k}))\ )$$

We can then consider the following program:

$$p_{cps} \triangleq \lambda\mathbf{k}.(\texttt{flip}_{cps}\ (\lambda\mathbf{x}.\lambda\mathbf{b}.\texttt{return}\ (\mathbf{k}\ \mathbf{x}))\ (\lambda\mathbf{k}.\lambda\mathbf{b}.(\mathbf{k}\ \mathbf{b})\ \mathbf{b})\ (\lambda(\mathbf{x},\mathbf{k}).\lambda\mathbf{b}.(\mathbf{k}\ ())\ \mathbf{x})\ )$$

In the rest of this section, we will aim to obtain this program from a suitable input.

Question #12

To which handler is this program equivalent ? Justify informally.

3.1 Code without closure

To make our task initially simpler, we consider a sublanguage of $\lambda_{\mathbf{eff}}$ without λ-expressions and function calls, named $\lambda^{-}_{\mathbf{eff}}$. The function $[\![e]\!]^H_k$ compiles the $\lambda^{-}_{\mathbf{eff}}$ expression e to Continuation Passing Style given a continuation k and a (meta) handler H. It is shown in Figure 7. The idea is, as we progress through handlers, to map each label `op` used to designate an algebraic effect with a continuation that indicates the code to execute when this effect is performed. H is thus a map from labels `op` to their continuations $k_{\mathtt{op}}$. Furthermore, k is a regular function symbol which indicates the current continuation.

$$\begin{array}{rll} H ::= & \{\mathtt{op} \mapsto k\} & \text{Effect continuation} \\ \mid & H \cup H' & \text{Union of handlers} \end{array}$$

Figure 6: Handler Continuations

$$\begin{aligned}
[\![\mathbf{x}]\!]^H_k &= \mathbf{x} \\
[\![v]\!]^H_k &= v \qquad \text{where } v \in \{(), \#t, \#f\} \cup \mathbb{Z} \\
[\![\mathtt{return}\ v]\!]^H_k &= k\ [\![v]\!]^H_k \\
[\![\mathtt{if}\ v\ \{\ c_1\ \}\ \mathtt{else}\ \{\ c_2\ \}]\!]^H_k &= \mathtt{if}\ v\ \{\ [\![c_1]\!]^H_k\ \}\ \mathtt{else}\ \{\ [\![c_2]\!]^H_k\ \} \\
[\![\mathtt{let}\ \mathbf{x} = c_1\ \mathtt{in}\ c_2]\!]^H_k &= ??? \\
[\![\mathtt{perform}\ op(v)]\!]^H_k &= k_{\mathtt{op}}\ v \qquad \text{where } \{\mathtt{op} \mapsto k_{\mathtt{op}}\} \in H \\
\left[\!\!\left[\mathtt{with}\ \left\{\begin{array}{l}\mathtt{return}\ \mathbf{x} \mapsto c_r, \\ op_i(\mathbf{x}, \mathbf{k}) \mapsto c_i, \quad \text{for } 0 \le i < n\end{array}\right\}\ \mathtt{handle}\ c\right]\!\!\right]^H_k &= \begin{array}{l}\mathtt{let}\ \mathbf{k_r} = \lambda \mathbf{x}.\ [\![c_r]\!]^H_k\ \mathtt{in} \\ \mathtt{let}\ \mathbf{k_{op_i}} = \lambda \mathbf{x}.\lambda \mathbf{k}.\ [\![c_i]\!]^H_k\ \mathtt{in} \qquad \text{for } 0 \le i < n \\ [\![c]\!]^{H \cup \{op_i \mapsto \mathbf{k_{op_i}} \mid 0 \le i < n\}}_{\mathbf{k_r}}\end{array}
\end{aligned}$$

Figure 7: Compilation of effects to CPS.

Question #13

Translate c_{ex} to CPS, and execute it with an initial cell set to true.

Question #14

In the translation of $\lambda^{-}_{\mathbf{eff}}$, do the meta handlers H appear at runtime? Explain their use.

Question #15

Give a rule for $[\![\mathtt{let}\ x = c_1\ \mathtt{in}\ c_2]\!]^H_k$ without using `let`. Where should k be used ? Justify the correctness informally.

Question #16

State (without proving) the correctness criterion for the compilation function $[\![e]\!]^H_k$.

Hint: this doesn't talk about soundness of the source language!

Question #17

With $\lambda^{-}_{\mathbf{eff}}$, we have neglected lambda-expressions. What is the problem with extending our compilation function to lambda-expressions? Give an example to illustrate the issue.

3.2 Implementing lambdas with dynamic handlers

As a first approach to handle lambda-expressions, we now introduce a new class of values: dynamic handlers! Similarly to before, handlers are maps from operations to continuations. But additionally, they can be bound to variables and composed. For instance,

$$\lambda \mathbf{x}.(\lambda \mathbf{H}.\texttt{let } \mathbf{H}' = \mathbf{H} \cup \{incr \mapsto c_{incr}\} \texttt{ in } \mathbf{H}'(get) + \mathbf{H}'(incr)\ 3)$$

The extended grammars and rules are given below. The idea is that all lambda-expressions now take two additional arguments: the current continuation, and the handler continuations.

$$\begin{array}{rl} v ::= \cdots \mid & H \quad \text{Dynamic Handler Continuations} \\ H ::= \cdots \mid & \mathbf{H} \quad\quad\quad\quad\quad\quad\quad\quad \text{Handler Variables} \end{array}$$

Figure 8: Dynamic Handler Continuations

$$\begin{aligned} [\![\lambda \mathbf{x}.c]\!]^H_k &= \lambda \mathbf{x}.\lambda \mathbf{k}'.\lambda \mathbf{H}'.\, [\![c]\!]^{\mathbf{H}'}_{\mathbf{k}'} \\ [\![v_1\ v_2]\!]^H_k &= ??? \\ [\![\texttt{perform } op(v)]\!]^H_k &= ??? \end{aligned}$$

Figure 9: Compilation rules, extended for lambda-expressions.

Question #18

Give the rule for application. Justify it informally.

Question #19

We have extended the notion of handler to be used dynamically.

Provide the new evaluation rules and extend the rule for `perform`. What changes compared to the previous rule ?

Question #20

Translate $\texttt{let } f = \lambda().\texttt{flip in with } h_{cell} \texttt{ handle } f()$ with this new technique.

Question #21

Prove your correctness theorem only on the "$\texttt{with } h \texttt{ handle } c$" case.

Question #22

Consider the following term.

$$\texttt{let } f = \lambda \mathbf{x}.(\texttt{with } \{rand() \mapsto c_{rand}\} \texttt{ handle perform set}(\mathbf{x});;\texttt{perform rand}()) \texttt{ in}$$
$$\texttt{with } h_{cell} \texttt{ handle } f\ 12$$

Translate it, and execute it. How many time is the operation of handlers union executed ?

Question #23

Propose a code simplification to reduce the number of handlers union.

Question #24

Consider a program without any effects. What performance penalties does our compilation to CPS incur ?

3.3 Optimising lambdas with an effect analysis

We now want to analyze a closure to know which effect it can raise, and use this information to optimise the compilation of lambdas.

Let us consider a judgement $\text{Effects}(c) = \texttt{op}_0, \ldots, \texttt{op}_{n-1}$ which indicates that the execution of c can only perform effects amongst $\texttt{op}_0, \ldots, \texttt{op}_{n-1}$. We also consider $\text{Effects}(v) = \xrightarrow{\texttt{op}_0,\ldots,\texttt{op}_{n-1}}$ which indicates that v is a function which, when called, can only perform effects amongst $\texttt{op}_0, \ldots, \texttt{op}_{n-1}$ (no matter what argument the function is called on).

For instance $\text{Effects}(\texttt{flip}) = \{\texttt{get;put}\}$

Question #25

Propose new compilation rules using this judgement to avoid translating effect-less functions to CPS. Justify their correctness.

Question #26 (Difficult)

Propose compilation rules that avoid the use of dynamic handler completely. Justify their correctness.

Question #27 (Difficult)

Open question

What are the techniques that could be used to implement $\text{Effects}(c) = \ldots$. Propose ideas.