Homework (DM)
Compilation and Program Analysis (CAP)

Code generation for a stream language

*Refer to the semantic course of CAP (chapter 03) for the semantic of **WHILE***

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.

Solution: These are "elements of solutions". typos, and precisions

Intro

Streams represent a (potentially unbounded) sequences of elements. In this homework, we consider a language called **STREAM**, which manipulates such streams using high level operations similar to functional programming languages. We will first define this language formally, and compile it to the imperative **WHILE** language seen during the course.

Syntax of the language is given in Figure 2. Example of **STREAM** programs are shown in Figure 1. A stream can be created "in extenso", through the syntax $[\![e_1..e_n]\!]$, or programmatically, via the `init` operation which takes a function f, a bound n, and returns the stream $[\![f(0), f(1), \ldots, f(n)]\!]$. Streams can be transformed through several operations such as `map` (transforms a stream with a function), `filter` (keeps only the elements respecting a predicate), `zip` (combines two streams), and `concatmap` (maps each element to a stream, and concatenates the

results). Finally, a stream can be collapsed into a single value using `fold`. A program is composed of a `fold` on an arbitrary stream expression composed of any of the previous operations. Single values can be integers and booleans, and we assume a language of arithmetic and boolean expressions similar to the one in the **WHILE** language. We use $x \to e$ to define a function that has a formal parameter x and a body e. We denote $e[x \mapsto e']$ the substitution of x by e' in e.

The semantics of **STREAM** is shown in Figure 3. We say that a program p (resp. a stream expression s) reduces to a value v, denoted $p \Downarrow v$. We also assume a semantic for arithmetic and boolean expressions similar to the one in the **WHILE** language.

$$\texttt{fold}(st \to x \to st + x,\ 0,\ \texttt{map}(i \to i * i,\ [\![0 \dots n]\!]))$$

(a) A simple **STREAM** program

$$\texttt{concatmap}(i \to \texttt{init}(j \to \binom{i}{j},\ i),\ [\![0 \dots n]\!])$$

(b) A **STREAM** expression with `concatmap`. $\binom{i}{j}$ is the binomial coefficient. The expression computes the elements of Pascal's triangle.

$$\begin{aligned}&\texttt{zip}(i \to j \to i + j,\\ &\quad \texttt{filter}(k \to k\%2 = 0,\ [\![0 \dots n]\!]),\\ &\quad \texttt{filter}(k \to k\%3 = 0,\ [\![0 \dots n]\!]))\end{aligned}$$

(c) A **STREAM** expression with `zip`

FIGURE 1 – Examples of **STREAM** programs

Stream Expressions

$$\begin{aligned} s ::= {} & [\![e_0, \dots, e_n]\!] \\ \mid {} & \texttt{init}(x \to e,\ e) \\ \mid {} & \texttt{map}(x \to e,\ s) \\ \mid {} & \texttt{filter}(x \to e,\ s) \\ \mid {} & \texttt{concatmap}(x \to s,\ s) \\ \mid {} & \texttt{zip}(x \to x' \to e,\ s,\ s') \end{aligned}$$

Programs

$$p ::= \texttt{fold}(x \to x' \to e,\ e_z,\ s)$$

Arithmetic and Boolean Expressions

$$\begin{aligned} v ::= {} & n \in \mathbb{N} \mid b \in \mathbb{B} \\ e ::= {} & x, y && \text{(Variables)} \\ \mid {} & v && \text{(Values)} \\ \mid {} & e + e' \mid e * e' \mid \dots && \text{(Arithmetic operations)} \\ \mid {} & e = e' \mid e < e' \mid \dots && \text{(Boolean operations)} \end{aligned}$$

Types

$$\begin{aligned} \tau ::= {} & \mathit{Int} \mid \mathit{Bool} && \text{(Base types)} \\ \mid {} & [\![\tau]\!] && \text{(Stream types)} \end{aligned}$$

FIGURE 2 – Grammar of the **STREAM** language

1 Typing and semantics

Question #1

What is the result of Figure 1a ? Show the derivation tree that allows you to obtain it.

Question #2

What does the `zip` operation do ? What happens if the two stream arguments are not of the same size ? Explain Figure 1c.

$$\frac{\forall i \in \{0 \dots n\},\ e_i \Downarrow v_i}{[\![e_0 \dots e_n]\!] \Downarrow [\![v_0 \dots v_n]\!]} \qquad \frac{s \Downarrow [\![v_0 \dots v_n]\!] \qquad \{j_0 \dots j_k\} = \{i \mid e[x \mapsto v_i] \Downarrow \texttt{true}\}}{\texttt{filter}(x \to e,\ s) \Downarrow [\![v_{j_0} \dots v_{j_k}]\!]}$$

$$\frac{e_n \Downarrow n \qquad \forall i \in \{0 \dots n\},\ e[x \mapsto i] \Downarrow v_i}{\texttt{init}(x \to e,\ e_n) \Downarrow [\![v_0 \dots v_n]\!]} \qquad \frac{s \Downarrow [\![v_0 \dots v_n]\!] \qquad \forall i \in \{0 \dots n\},\ e[x \mapsto v_i] \Downarrow v'_i}{\texttt{map}(x \to e,\ s) \Downarrow [\![v'_0 \dots v'_n]\!]}$$

$$\frac{s' \Downarrow [\![v_0 \dots v_n]\!] \qquad \forall i \in \{0 \dots n\},\ s[x \mapsto v_i] \Downarrow \left[\!\!\left[v'_{i,0} \dots v'_{i,k_i}\right]\!\!\right]}{\texttt{concatmap}(x \to s,\ s') \Downarrow [\![v'_{0,0} \dots v'_{0,k_0} v'_{1,0} \dots v'_{n,k_n}]\!]}$$

$$\frac{s \Downarrow [\![v_0 \dots v_n]\!] \qquad s' \Downarrow [\![v'_0 \dots v'_{n'}]\!] \qquad m = min(n, n') \qquad \forall i \in \{0 \dots m\},\ e[x \mapsto v_i, y \mapsto v'_i] \Downarrow v''_i}{\texttt{zip}(x \to y \to e,\ s,\ s') \Downarrow [\![v''_0 \dots v''_m]\!]}$$

$$\frac{s \Downarrow [\![v_0 \dots v_n]\!] \qquad e_z \Downarrow st_0 \qquad \forall i \in \{0 \dots n\},\ e[x \mapsto st_i, y \mapsto v_i] \Downarrow st_{i+1}}{\texttt{fold}(x \to y \to e,\ e_z,\ s) \Downarrow st_{n+1}}$$

FIGURE 3 – Semantic of the **STREAM** language

Given Γ is the typing environment, a is a stream expression and τ a type, we consider the typing judgement $\Gamma \vdash s : \tau$, read as "s has type τ in Γ".
The typing rule for `fold` is shown below ($\mapsto$ is the mapping update)

$$\frac{\Gamma \vdash e_z : \tau_r \qquad \Gamma \vdash s : [\![\tau]\!] \qquad \Gamma[st \mapsto \tau_r, x \mapsto \tau] \vdash e : \tau_r}{\Gamma \vdash \texttt{fold}(st \to x \to e,\ e_z,\ s) : \tau_r}$$

For boolean and arithmetic expressions, we assume the typing is similar to the one of the **WHILE** language.

Question #3

Write the typing rule for the remaining combinators of **STREAM** : `map`, `filter`, `zip` and `concatmap`.

Solution: TODO

Question #4

Show the typing derivation of the example in Figure 1a.

Solution: TODO

We now state the preservation theorem for the **STREAM** language.

Theorem 1 (Preservation) *Let Γ be an environment, e a stream expression, τ a type, and c a value.*

$$\Gamma \vdash e : \tau \land e \Downarrow c \implies \Gamma \vdash c : \tau$$

Question #5

(a) Explain this theorem : what does it guarantee ?

(b) How is this kind of theorem proved (notably, for the **WHILE** language during the course) ?

(c) What other results are proved about the **WHILE** language ? Can we formulate them here ?

Question #6

Prove preservation for the case `filter` and `concatmap`.

Solution: TODO

Question #7

We now consider a new "`mapn`" operation, which generalizes the `zip` (and the `map`) operation to n arguments. It takes a function of n arguments and n streams, and apply the function simultaneously on all streams. We add the new syntax $\texttt{mapn}(x_0 \dots x_n \to s, s_0, \dots, s_n)$ to stream expressions s.

Write the semantic and the typing rule for `mapn`. Be mindful of the number of arguments.

Termination and worst case execution

Notice that all the initial streams (literal and init) of the language are bounded.

Question #8

For each stream operation, show that if input streams are finite, the output stream is finite as well.

Question #9

Prove that every well typed **STREAM** program terminates.

Question #10 (Bonus)

Is it possible to give an upper bound on the size of the stream produced by an expression ?

Question #11 (Bonus)

What can you say about the semantics if we add an unbounded init function which creates an infinite stream.

2 Compilation

We now compile **STREAM** to a subset of the **WHILE** language. Before doing so, we want to simplify our programs using fusion rules.

Fusion

Fusion rules are used to merge operations used in sequence. For instance, we can consider the following rule on `map` operations :

$$\texttt{map}(x' \to e',\ \texttt{map}(x \to e,\ s)) \longrightarrow \texttt{map}(x \to e'[x' \mapsto e],\ s)$$

Question #12

Show that this rule preserves both typing and reduction : if the expression before fusion is well-typed then the expression after fusion is well-typed ; and if the expression before fusion reduces to a stream then the expression after fusion reduces to the same stream.

Question #13

Show that both `map` and `filter` can be expressed in term of `concatmap`.

Question #14

Show a rule that merges successive uses of `concatmap` and prove that it preserves typing and reduction.

Question #15

Can `zip` also be expressed in term of `concatmap` ? Why ? Is there a rule to merge `zip` and `concatmap` operations ?

Which operator would generalize `zip` and `concatmap` (and allow fusion rules between `zip` and `concatmap`) ?

Compilation to WHILE

Thanks to the rules previously shown, we have simplified our language to only contains literal streams and `concatmap`, `zip` and `fold` operations. We are now ready to compile to **WHILE** programs. We will equip ourselves with additional functions to manipulate streams :

— $[\![\ldots]\!]$ is a stream literal.

— $\texttt{next}(x)$ returns the next element of the stream x.

— $\texttt{hasnext}(x)$ returns `true` iff the stream x is not empty.

— $\texttt{push}(x, v)$ pushes the value v at the end of the stream x.

The compiled program will assign the computed result of the final fold operation in a predefined variable *res*. We use 0 as the initial value for variables regardless of their type ()

$$
\begin{aligned}
&\textsf{var } a : [\![Int]\!] = [\![0 \ldots n]\!]\,; \\
&\textsf{var } b : [\![Int]\!] = [\![]\!]\,; \\
&\textsf{var } c : Int = 0; \\
&\textsf{var } res : Int = 0; \\
&\texttt{while}(\texttt{hasnext}(a))\{c := \texttt{next}(a); \texttt{push}(b, c * c)\}; \\
&\texttt{while}(\texttt{hasnext}(b))\{res := res + \texttt{next}(b)\}
\end{aligned}
$$

FIGURE 4 – A compiled **STREAM** program

Question #16

Justify informally that the program in Figure 4 has the same semantics as Figure 1a.

We note Compile(s, x) the code computing s and storing the result in x. We define the compilation of `fold` as follow :

$$\text{Compile}(\texttt{fold}(x \to y \to e,\ e_z,\ s), t) = \left|\begin{array}{l} D; \\ \texttt{var } t : Int = 0; \\ \texttt{var } w : Int = 0; \\ \texttt{Body}; \\ t := e_z; \\ \texttt{while}(\texttt{hasnext}(u))\{ \\ \quad w := \texttt{next}(u); \\ \quad t := e[x \mapsto t, y \mapsto w]; \\ \} \end{array}\right. \quad \begin{array}{l} \text{where } u, w \text{ are fresh variables} \\ \text{and Compile(s,u) = D ;Body} \end{array}$$

Question #17

Give the remaining compilation rules. You only need to provide rules for the `concatmap` and `zip` operations.

Question #18 (Bonus)

Justify informally the correctness of your compilation rules.