Homework (DM)
Compilation and Program Analysis (CAP)

Type-directed compilation for a dynamic 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

In this homework, we consider a language called **PIE**, a dynamically typed Python-like programming language which contains operator such as + working on integers, arrays, booleans, ... For performance reasons, we want to compile **PIE** to **WHILE**, a statically typed C-like language. Our compilation procedure will therefor add dynamic type tests, manifested by `is_int` or `is_list` functions in the generated **WHILE** code. Such tests are generally costly. In a second step, we will define a typing analysis and a type-directed compilation step to avoid such tests when they are not needed.

We now define **PIE** formally. Syntax of the language is given in Figure 2. Example of **PIE** programs are shown in Figure 1. The language follows a python-like syntax with booleans,

integers, and lists of integers. Statements includes assignments, if-then-else, and while loops. Finally, expressions can be constants, equality test or "addition". Additional works on any values of the same type: `3 + 5 = 8`, `[1] + [2,3] = [1,2,3]`, and `true + false = true`. Additionally, if-then-else expressions accept any value as test, and will test for "truthyness". For instance, Figure 1a is a valid program, and returns 42. Finally, the last line in Figure 1c can be either an addition on integer, or a concatenation of lists, depending on the previous execution.

```
l:=3
if l:
  r:=42
else:
  r:=0
```

(a) A test on an integer

```
if l:
  x,l:=pop(l)
else:
  x:=0
```

(b) A piece of code using pop

```
if b:
  x:=3
  y:=4
else:
  x:=[1]
  y:=[2]
z:=x+y
```

(c) A dynamically typed addition

Figure 1: Examples of **PIE** programs

Expressions:

$$
\begin{array}{rll}
e \in \mathcal{E} ::= & \texttt{true} \mid \texttt{false} & \\
\mid & n \mid \texttt{[]} & \text{Constants} \\
\mid & \texttt{[}e\texttt{]} & \text{Singletons} \\
\mid & x & \text{Variable} \\
\mid & (e == e') & \text{Equality} \\
\mid & e + e & \text{Addition}
\end{array}
$$

Statements:

$$
\begin{array}{rll}
S \in Stm ::= & (x := e) & \text{Assign} \\
\mid & x, y := pop(e) & \text{Pop} \\
\mid & skip & \text{Do nothing} \\
\mid & S_1; S_2 & \text{Sequence} \\
\mid & \texttt{if } e \texttt{ :} \; S_1 \; \texttt{else :} \; S_2 & \text{Test} \\
\mid & \texttt{while } e \texttt{ :} \; S & \text{Loop}
\end{array}
$$

Figure 2: Grammar of the **PIE** language

The semantics of **PIE** is shown in Figure 3. The domain of values contains the traditional integers $\mathbb{Z}$ and booleans $\mathbb{B}$, along with lists of integers $\mathbb{L} = \mathbb{Z}^n$. We note $+\!+$ the concatenation on lists.

We then define the evaluation of expressions, noted *Val*, taking an expression and a variable environment. The only peculiar behavior is the evaluation of $e_1 + e_2$, which uses the dynamically typed behavior highlighted before. We define truthyness as a function from values to booleans which analyze the value dynamically, and decide if it's truthy or not. Finally, We define the evaluation of statement using a small step semantics. Again, the main pecularity is the evaluation of the if-then-else statement, which uses truthyness.

Value domain

$$\mathbb{L} = \mathbb{Z}^n \quad \text{(List of } \mathbb{Z}\text{)}$$
$$\mathbb{V} = \mathbb{Z} \cup \mathbb{B} \cup \mathbb{L} \cup \{\bot\} \quad \text{(Values)}$$
$$State = Var \to \mathbb{Z} \cup \mathbb{B} \cup \mathbb{L}$$

Truthyness

$$\texttt{is\_truthy} : \mathbb{V} \to \mathbb{B} \cup \{\bot\}$$
$$\texttt{is\_truthy}(0) = \texttt{false}$$
$$\texttt{is\_truthy}(\texttt{false}) = \texttt{false}$$
$$\texttt{is\_truthy}(\texttt{[]}) = \texttt{false}$$
$$\texttt{is\_truthy}(\bot) = \bot$$
$$\texttt{is\_truthy}(v) = \texttt{true} \qquad \text{otherwise}$$

Evaluation of Expressions

$$Val : \mathcal{E} \times State \to \mathbb{V}$$
$$Val(n, \sigma) = value(n)$$
$$Val(x, \sigma) = \sigma(x)$$
$$Val(e_1 + e_2, \sigma) = \begin{cases} v_1 + v_2 & \text{if } v_1, v_2 \in \mathbb{Z} \\ v_1 \text{ or } v_2 & \text{if } v_1, v_2 \in \mathbb{B} \\ concat(v_1, v_2) & \text{if } v_1, v_2 \in \mathbb{L} \\ \bot & \text{otherwise} \end{cases}$$
where $v_1 = Val(e_1, \sigma)$ and $v_2 = Val(e_2, \sigma)$

Evaluation of Statements $(Stm, State) \Rightarrow (Stm, State)$ or $(Stm, State) \Rightarrow State$

$$\frac{Val(e,\sigma) \neq \bot}{(x := e, \sigma) \Rightarrow \sigma[x \mapsto Val(e,\sigma)]} \qquad \frac{Val(e,\sigma) = [v] + +L}{(x, y := pop(e), \sigma) \Rightarrow \sigma[x \mapsto v][y \mapsto L]} \qquad (\texttt{skip}, \sigma) \Rightarrow \sigma$$

$$\frac{(S_1, \sigma) \Rightarrow \sigma'}{((S_1; S_2), \sigma) \Rightarrow (S_2, \sigma')} \qquad \frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{((S_1; S_2), \sigma) \Rightarrow (S_1'; S_2, \sigma')} \qquad \frac{\texttt{is\_truthy}(Val(b, \sigma)) = \texttt{true}}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_1, \sigma)}$$

$$\frac{\texttt{is\_truthy}(Val(b, \sigma)) = \texttt{false}}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_2, \sigma)}$$

$$(\texttt{while } b \texttt{ do } S \texttt{ done}, \sigma) \Rightarrow (\texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S \texttt{ done}) \texttt{ else skip}, \sigma)$$

Figure 3: Semantic of the **PIE** language

1 Semantics

Question #1

Explain the difference between big and small step semantics

Solution: See course.

Question #2

Give an example of program where the conditional in `while` is not a boolean.

Give an example of program where the conditional in `while` changes types across execution of the loop.

Solution:

```
i := 5
while (i):
  if (i > 100):
    i := false;
  else:
    i := (i+2)/3
```

1.1 The WHILE language

We present below the syntax of **WHILE** as seen in the course The syntax and rules are the same as present in the course, with some additions built-in operations for dynamic tests and lists.

The grammar is shown in Figure 4. Expressions have been extended with lists of integer including the empty list `[]`, the list with a single element `[`n`]`, and a concatenation operator. We also have operators `is_int`, `is_bool` and `is_list` to check the type of an expression.

The semantics is shown in Figure 5 and follows the one given in the course. No operator is available for mixed types: addition only works with two integers, concatenation $+\!\!+$ with two lists and equality $==$ for any two expressions of the same type.

Expressions:

$$
\begin{array}{rll}
e \in \mathcal{E}_w ::= & \texttt{true} \mid \texttt{false} \mid n \mid \texttt{[]} & \text{Constants} \\
\mid & \texttt{[}e\texttt{]} & \text{Singletons} \\
\mid & x & \text{Variable} \\
\mid & (b \mid\mid b) \mid (b \;\&\&\; b) & \text{Bool Op} \\
\mid & (e == e') & \text{Equality} \\
\mid & e + e & \text{Addition} \\
\mid & \texttt{concat}(e, e') & \text{List concat} \\
\mid & \texttt{is\_int} \mid \texttt{is\_bool} \mid \texttt{is\_list} & \text{Type checks}
\end{array}
$$

Statements:

$$
\begin{array}{rll}
S \in Stm_w ::= & (x := e) & \text{Assign} \\
\mid & x, y := pop(e) & \text{Pop} \\
\mid & skip & \text{Do nothing} \\
\mid & S_1; S_2 & \text{Sequence} \\
\mid & \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 & \text{Test} \\
\mid & \texttt{while } e \texttt{ do } S \texttt{ done} & \text{Loop} \\
\mid & \texttt{Fail} & \text{Fail}
\end{array}
$$

Figure 4: Grammar of the **WHILE** language

Evaluation of Expressions: $Val : \mathcal{E} \times State \to \mathbb{Z} \cup \mathbb{B} \cup \mathbb{L}$

$$
\begin{array}{ll}
Val(n, \sigma) = value(n) & Val(\texttt{is\_int}(e), \sigma) = \texttt{true} \quad \text{if } Val(e, \sigma) \in \mathbb{Z} \\
Val(x, \sigma) = \sigma(x) & Val(\texttt{is\_int}(e), \sigma) = \texttt{false} \quad \text{otherwise} \\
Val(e_1 + e_2, \sigma) = Val(e_1, \sigma) + Val(e_2, \sigma) & Val(\texttt{is\_list}(e), \sigma) = \texttt{true} \quad \text{if } Val(e, \sigma) \in \mathbb{L} \\
Val(concat(e_1, e_2), \sigma) = Val(e_1, \sigma) +\!\!+ Val(e_2, \sigma) & Val(\texttt{is\_list}(e), \sigma) = \texttt{false} \quad \text{otherwise} \\
\ldots & \ldots
\end{array}
$$

Evaluation of Statements $(Stm, State) \Rightarrow (Stm, State) \mid State \mid Error$

$$(x := e, \sigma) \Rightarrow \sigma[x \mapsto Val(e, \sigma)] \qquad (\texttt{skip}, \sigma) \Rightarrow \sigma \qquad (\texttt{Fail}, \sigma) \Rightarrow Error$$

$$\frac{Val(e, \sigma) = \texttt{[}v\texttt{]} +\!\!+ L}{(x, y := \texttt{pop}(e), \sigma) \Rightarrow \sigma[x \mapsto v][y \mapsto L]} \qquad \frac{(S_1, \sigma) \Rightarrow \sigma'}{((S_1; S_2), \sigma) \Rightarrow (S_2, \sigma')} \qquad \frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{((S_1; S_2), \sigma) \Rightarrow (S_1'; S_2, \sigma')}$$

$$\frac{(S_1, \sigma) \Rightarrow Error}{((S_1; S_2), \sigma) \Rightarrow Error} \qquad \frac{Val(b, \sigma) = \texttt{true}}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_1, \sigma)}$$

$$\frac{Val(b, \sigma) = \texttt{false}}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_2, \sigma)}$$

$$(\texttt{while } b \texttt{ do } S \texttt{ done}, \sigma) \Rightarrow (\texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S \texttt{ done}) \texttt{ else skip}, \sigma)$$

Figure 5: Semantic of the **WHILE** language

Question #3

Write a program that does the same computation as the last one you provided in Question #2, where the conditional in `while` changes types across execution of the loop. Note that while in **WHILE** only accepts boolean conditions. You can introduce additional variables.

Solution:

```
i := 5
b := 5
while (not (i == 0 || b)) {
  if (i > 100) then
    b := false;
  else
    i := (i+2)/3
}
```

2 Naive Compilation

We now compile **PIE** to the **WHILE** language. We recall that our **WHILE** language possess three extra functions: `is_bool`, `is_int` and `is_list`. Given p a **PIE** program, we note $[\![p]\!]$ the corresponding **WHILE** program.

2.1 Compilation of expressions

We first look at expressions. Given an **PIE** expression e, $[\![e]\!]$ returns a pair composed of a **WHILE** statement S_w and a variable x_w, such that x_w is defined after the execution of S_w, and contains the returned value computed by e.

$$\frac{}{[\![x]\!] = skip, x} \qquad \frac{x \text{ fresh}}{[\![n]\!] = (x := n), x} \qquad \frac{[\![e]\!] = S, x \qquad x_L \text{ fresh}}{[\![[e]]\!] = (S; x_L := [x]), x_L}$$

$$\frac{[\![e_1]\!] = S_1, x_1 \qquad [\![e_2]\!] = S_2, x_2 \qquad x_b \text{ fresh}}{[\![e_1 == e_2]\!] = (S_1; S_2; x_b := (x_1 == x_2)), x_b} \qquad \frac{\text{TO COMPLETE} \qquad x_f \text{ fresh}}{[\![e + e']\!] = \text{TO COMPLETE}, x_f}$$

Question #4

Complete the compilation of expressions by defining the compilation of $+$.

Solution:

$$\frac{[\![e]\!] = S_1, x_1 \qquad [\![e']\!] = S_2, x_2 \qquad x_f \text{ fresh}}{[\![e + e']\!] = \begin{array}{l} S_1; S_2; \\ \texttt{if } (\texttt{is\_bool}(x_1) \,\&\&\, \texttt{is\_bool}(x_2))\{x_f := x_1 \,||\, x_2\} \\ \texttt{else if } (\texttt{is\_int}(x_1) \,\&\&\, \texttt{is\_int}(x_2))\{x_f := x_1 + x_2\} \\ \texttt{else if } (\texttt{is\_list}(x_1) \,\&\&\, \texttt{is\_list}(x_2))\{x_f := concat(x_1, x_2)\} \\ \texttt{else } \{\texttt{Fail}\} \end{array}, x_f$$

2.2 Compilation of statements

We now define the compilation of statements. Given a **PIE** statement S, $[\![S]\!]$ returns a **WHILE** statement S_w. We use an auxiliary function $Truthy(x)$ which outputs a statement S_w and a variable x_w which indicates if x is truthy.

$$\frac{x_w \text{ fresh}}{Truthy(x) = \begin{array}{l}\texttt{if (is\_bool}(x)\texttt{)}\{x_w := x\}\\ \texttt{else if (is\_int}(x)\texttt{ \&\& } x == 0\texttt{)}\{x_w := \texttt{false}\}\\ \texttt{else if (is\_list}(x)\texttt{ \&\& } x == \texttt{[]}\texttt{)}\{x_w := \texttt{false}\}\\ \texttt{else } \{x_w := \texttt{true}\}\end{array}, x_w}$$

$$\frac{}{[\![skip]\!] = skip} \qquad \frac{[\![e]\!] = S_w, x_w}{[\![x := e]\!] = S_w; x := x_w}$$

$$\frac{[\![e]\!] = S_w, x_w \qquad S_{test}, x_{test} = Truthy(x_w) \qquad [\![S_0]\!] = S_{w,0} \qquad [\![S_1]\!] = S_{w,1}}{\left[\!\!\left[\begin{array}{l}\texttt{if } e:\\ \quad S_0\\ \texttt{else}:\\ \quad S_1\end{array}\right]\!\!\right] = \begin{array}{l}S_w;\\ S_{test};\\ \texttt{if } x_{test} \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}\end{array}}$$

$$\frac{\text{TO COMPLETE}}{[\![x, l := pop(e)]\!] = \text{TO COMPLETE}} \qquad \frac{\text{TO COMPLETE}}{\left[\!\!\left[\begin{array}{l}\texttt{while } e:\\ \quad S\end{array}\right]\!\!\right] = \text{TO COMPLETE}}$$

Question #5

Define the compilation of pop.

Solution:

$$\frac{[\![e]\!] = S_w, x_w}{[\![x, l := pop(e)]\!] = S_w;\ x, l := pop(x_w)}$$

Question #6

Compile the code of Figure 1b.

Solution:

```
if (is_bool(l)){ b := l }
else if (is_int(l) && l == 0){b := false}
```

```
else if (is_list(l) && l == []){b := false}
else {b := true}
if (b) then
  x, l := pop(l)
else
  x := 0
```

Question #7 (Difficult)

Define the compilation of `while`. You can use *Truthy*.

Solution:

$$\frac{[\![e]\!] = S_w, x_w \qquad S_{test}, x_{test} = Truthy(x_w) \qquad [\![S]\!] = S_body}{\left[\!\left[\begin{array}{l}\texttt{while } e : \\ \quad S\end{array}\right]\!\right] = S_w; S_{test}; \texttt{while } x_{test} \texttt{ do } S_{body}; S_w; S_{test} \texttt{ done}}$$

We now want to prove the correction of the compilation step.

Question #8 (Difficult)

Prove the correctness of expression compilation. Formally you should prove the following statement:

$$\begin{cases} [\![e]\!] = S_W, x_W \\ \sigma \subseteq \sigma_W \\ Val(\sigma, e) = v \neq \bot \end{cases} \implies \exists \sigma'_W. \begin{cases} \sigma \subseteq \sigma'_W \\ (S_W, \sigma_W) \Rightarrow^* \sigma'_W \\ Val(\sigma'_W, x_W) = v \end{cases}$$

The proof is an induction, you will only consider the following cases: integer constant n, singleton $[e]$, and $+$ operator $(e + e')$.

Solution: This is an induction on the syntax of expressions, we consider the case:

* $[\![n]\!] = x := n, x$ with $Val(\sigma, n) = value(n)$

Suppose we have σ_W verifying the left hand side hypothesis

We have that: $(x := n, \sigma_W) \Rightarrow \sigma_W[x \mapsto value(n)]$, which defines σ'_W, inclusion is ensured by freshness (this is trivial but should be formalized), finally $Val(\sigma'_W, x) = value(n)$ holds.

* $[\![\texttt{[}e\texttt{]}]\!]$ with

$$\frac{[\![e]\!] = S, x \qquad x_L \text{ fresh}}{[\![\texttt{[}e\texttt{]}]\!] = (S; x_L := \texttt{[}x\texttt{]}), x_L}$$

and $Val(\sigma, \texttt{[}e\texttt{]}) = [v]$ where $v = Val(\sigma, e)$ by definition of Val

by recurrence hypothesis applied on e we obtain a σ'_W such that

$$\begin{cases} \sigma \subseteq \sigma'_W \\ (S, \sigma_W) \Rightarrow^* \sigma'_W \\ Val(\sigma'_W, x) = v \end{cases}$$

We then have (small step for sequence, assignment and evaluation of list expressions)

$$(S; x_L := \texttt{[}x\texttt{]}), \sigma_W) \Rightarrow^* (x_L := \texttt{[}x\texttt{]}, \sigma'_W) \Rightarrow \sigma'_W[x_L \mapsto [Val(\sigma'_W, x)]$$

Which defines σ''_W such that

$$\begin{cases} \sigma \subseteq \sigma''_W & \text{by construction only fresh variables are modified} \\ (S, \sigma_W) \Rightarrow^* \sigma''_W & \text{see just above} \\ Val(\sigma''_W, x_L) = [v] & \text{simple composition of valuation results} \end{cases}$$

This concludes the second case

* $[\![e + e']\!]$ with

$$\frac{[\![e]\!] = S_1, x_1 \qquad [\![e']\!] = S_2, x_2 \qquad x_f \text{ fresh}}{[\![e + e']\!] = \begin{array}{l} S_1; S_2; \\ \texttt{if } (\texttt{is\_bool}(x_1) \,\&\&\, \texttt{is\_bool}(x_2))\{x_f := x_1 \parallel x_2\} \\ \texttt{else if } (\texttt{is\_int}(x_1) \,\&\&\, \texttt{is\_int}(x_2))\{x_f := x_1 + x_2\} \\ \texttt{else if } (\texttt{is\_list}(x_1) \,\&\&\, \texttt{is\_list}(x_2))\{x_f := concat(x_1, x_2)\} \\ \texttt{else } \{\texttt{Fail}\} \end{array}, x_f}$$

and $Val(\sigma, e + e') = Val(\sigma, e) + Val(\sigma, e') = v_1 + v_2$ by definition of expression evaluation (in Pie). σ_W is given by the hypothesis of the thm.

by recurrence hypothesis applied on e and σ_W (+ recall other hypotheses) we obtain a σ'_W such that

$$\begin{cases} \sigma \subseteq \sigma'_W \\ (S, \sigma_W) \Rightarrow^* \sigma'_W \\ Val(\sigma'_W, x) = v_1 \end{cases}$$

by recurrence hypothesis applied on e' and from σ'_W (+ recall other hypotheses, in particular it is crucial to note here that we use the fact that $\sigma \subseteq \sigma'_W$ to apply IH) we obtain a σ''_W such that

$$\begin{cases} \sigma \subseteq \sigma''_W \\ (S, \sigma_W) \Rightarrow^* \sigma''_W \\ Val(\sigma''_W, x) = v_2 \end{cases}$$

We then have (small step and results above)

$$\begin{array}{rl} & S_1; S_2; \\ & \texttt{if } (\texttt{is\_bool}(x_1) \,\&\&\, \texttt{is\_bool}(x_2))\{x_f := x_1 \parallel x_2\} \\ & (\texttt{else if } (\texttt{is\_int}(x_1) \,\&\&\, \texttt{is\_int}(x_2))\{x_f := x_1 + x_2\} \qquad , \sigma_W) \\ & \texttt{else if } (\texttt{is\_list}(x_1) \,\&\&\, \texttt{is\_list}(x_2))\{x_f := concat(x_1, x_2)\} \\ & \texttt{else } \{\texttt{Fail}\} \\ & \texttt{if } (\texttt{is\_bool}(x_1) \,\&\&\, \texttt{is\_bool}(x_2))\{x_f := x_1 \parallel x_2\} \\ \Rightarrow * (& \texttt{else if } (\texttt{is\_int}(x_1) \,\&\&\, \texttt{is\_int}(x_2))\{x_f := x_1 + x_2\} \\ & \texttt{else if } (\texttt{is\_list}(x_1) \,\&\&\, \texttt{is\_list}(x_2))\{x_f := concat(x_1, x_2)\} \quad , \sigma''_W) \\ & \texttt{else } \{\texttt{Fail}\} \end{array}$$

Suppose that v_1 and v_2 are integers, we then have

$$\left(\begin{array}{l}\texttt{if } (\texttt{is\_bool}(x_1)\ \&\&\ \texttt{is\_bool}(x_2))\{x_f := x_1 \parallel x_2\}\\ \texttt{else if } (\texttt{is\_int}(x_1)\ \&\&\ \texttt{is\_int}(x_2))\{x_f := x_1 + x_2\}\\ \texttt{else if } (\texttt{is\_list}(x_1)\ \&\&\ \texttt{is\_list}(x_2))\{x_f := concat(x_1, x_2)\}\\ \texttt{else } \{\texttt{Fail}\}\end{array}, \sigma''_W\right) \Rightarrow^* \sigma''_W[x_f \mapsto v_1 + v_2]$$

which is sufficient to define σ'''_W ensuring the requirements for the proof. Cases where both elements are lists or bool are similar. Other cases fail (on both sides but not in the same way).

Question #9 (Difficult)

Prove the correctness of statement compilation. Namely prove the two following assertions:

$$\begin{cases}[\![S]\!] = S_W\\ \sigma \subseteq \sigma_W\\ (S, \sigma) \Rightarrow (S', \sigma')\end{cases} \implies \exists S'_W, \sigma'_W. \begin{cases}(S_W, \sigma_W) \Rightarrow^* (S'_W, \sigma'_W)\\ [\![S']\!] = S'_W\\ \sigma' \subseteq \sigma'_W\end{cases}$$

and

$$\begin{cases}[\![S]\!] = S_W\\ \sigma \subseteq \sigma_W\\ (S, \sigma) \Rightarrow \sigma'\end{cases} \implies \exists \sigma'_W. \begin{cases}(S_W, \sigma_W) \Rightarrow^* \sigma'_W\\ \sigma' \subseteq \sigma'_W\end{cases}$$

The proof is an induction, you will only consider the following cases: if statement and assignment.

Solution: By induction on the syntax.

* Case assign (we have σ_W, ...)

$$\frac{[\![e]\!] = S_w, x_w}{[\![x := e]\!] = S_w; x := x_w}$$

Let $v = Val(e, \sigma)$. We have:

$$(x := e, \sigma) \Rightarrow \sigma[x \mapsto v] = \sigma'$$

By Q8 we have

$$\begin{cases}\sigma \subseteq \sigma'_W\\ (S_W, \sigma_W) \Rightarrow^* \sigma'_W\\ Val(\sigma'_W, x_W) = v\end{cases}$$

and by SOS rule for sequence:

$$(S_w; x := x_w, \sigma_W) \Rightarrow^* (x := x_w, \sigma'_W)$$

Finally $(x := x_w, \sigma'_W) \Rightarrow \sigma'_W[x \mapsto Val(\sigma'_W, x_W)] = \sigma''_W \supseteq \sigma'$ This concludes for assignment.

* case if

$$\frac{[\![e]\!] = S_w, x_w \qquad S_{test}, x_{test} = Truthy(x_w) \qquad [\![S_0]\!] = S_{w,0} \qquad [\![S_1]\!] = S_{w,1}}{\left[\!\!\left[\begin{array}{l}\texttt{if } e:\\ \quad S_0\\ \texttt{else}:\\ \quad S_1\end{array}\right]\!\!\right] = \begin{array}{l} S_w;\\ S_{test};\\ \texttt{if } x_{test} \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}\end{array}}$$

We distinguish cases on $Val(e, \sigma)$. We detail the case `true`; other cases are similar Thus, suppose $Val(e, \sigma) = \texttt{true}$. We have:

$$(\begin{array}{l}\texttt{if } e:\\ \quad S_0\\ \texttt{else}:\\ \quad S_1\end{array}, \sigma) \Rightarrow (S_0, \sigma)$$

By Q8 we have

$$\begin{cases} \sigma \subseteq \sigma'_W \\ (S_W, \sigma_W) \Rightarrow^* \sigma'_W \\ Val(\sigma'_W, x_W) = \texttt{true} \end{cases}$$

and by SOS rule for sequence:

$$\begin{array}{rll} (S_w; S_{test}; \texttt{if } x_{test} \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}, \sigma_W) \Rightarrow^* & & \\ (S_{test}; \texttt{if } x_{test} \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}, \sigma'_W) \Rightarrow^* & & \text{see above} \\ (x_{test} := x_w; \texttt{if } x_{test} \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}, \sigma'_W) \Rightarrow^* & & \text{seq + isbool + ... inside } S_{test} \\ (\texttt{if } x_{test} \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}, \sigma'_W[x_{test} \mapsto \texttt{true}]) \Rightarrow^* & & \text{seq+assign} \\ (S_{w,0}, \sigma'_W[x_{test} \mapsto \texttt{true}]) & \sigma'_W(x_{test}) = \texttt{true}, & \text{SOS rule for if} \end{array}$$

where x_{test} is fresh and returned by truthy.

We can then conclude as $[\![S_0]\!] = S_{w,0}$, and $\sigma \subseteq \sigma'_W[x_{test} \mapsto \texttt{true}]$ by construction.

Question #10

We now try to investigate when the program compiled from an expression can lead to an error. Write the property that states that execution of a compiled expression only fails if execution of the source **PIE** expression also fails. In other words, given e a **PIE** expression, let $[\![e]\!] = S_W, x_W$, under which condition on $Val(e, \sigma)$ do we have $(S_W, \sigma_W) \Rightarrow^* Error$. Is the condition necessary and sufficient?

Note: you do not need to prove the property

Solution: Provided $[\![e]\!] = S, x \wedge \sigma \subseteq \sigma_W$ We have:
$Val(\sigma, e) = \bot \implies (S, \sigma_W) \Rightarrow^* Fail$

conversely we can state that if the reduction fails and it is a compilation of an expression,

the expression was invalid, The condition is necessary and sufficient.
(this supposes that the semantics we have defined for + ensures this.)

Question #11
Informally state **in one or two sentences** a similar property on statements.

Solution: Pie reduction not stuck implies the compiled version does not reduce to an error

3 Optimizing compilation

The code we emit so far is very naive: For instance, if we add two integers, we will always check their type before doing the addition. Similarly, if the condition of an `if` is always a boolean, we will still consider the list and integer cases.

We want to improve this by statically detecting if the types of operand are known, and remove redundant cases if possible. For this purpose, we first develop a type analysis that evaluate which type can appear at a specific point in the program. We then use this type information to emit optimized code through additional compilation rules.

3.1 Typing Analysis

We now define a typing analysis that tries to associate a set of potential types to each expression, and infers a typing environment that maps each variable to such set of types. We consider **B**, **Z** and **L** the type of booleans, integers and lists. We note the domain of types $\mathcal{T} = \mathbf{B} \mid \mathbf{Z} \mid \mathbf{L}$ and our typing environment $\Gamma = Var \to \mathcal{P}(\mathcal{T})$ from variables to sets of types.

We now define our analysis judgement on expressions, denoted $\Gamma \vdash \mathcal{E} : \mathcal{P}(\mathcal{T})$, which returns a set of types and on statements, denoted $\Gamma \vdash Stm : \Gamma$ which returns an environment. We rely on a set comparison operator: $\Gamma \subseteq \Gamma'$ is true if for each x, we have $\Gamma(x) \subseteq \Gamma'(x)$.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \{\mathbf{Z}\}} \qquad \frac{}{\Gamma \vdash \texttt{true/false} : \{\mathbf{B}\}} \qquad \frac{}{\Gamma \vdash [e] : \{\mathbf{L}\}}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 + e_2 : T_1 \cup T_2} \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash x := e : \Gamma[x \mapsto T]} \qquad \frac{\Gamma \vdash S_1 : \Gamma' \qquad \Gamma' \vdash S_2 : \Gamma''}{\Gamma \vdash S_1; S_2 : \Gamma''}$$

$$\frac{\Gamma \subseteq \Gamma_f \qquad \Gamma_f \vdash S : \Gamma' \qquad \Gamma' \subseteq \Gamma_f}{\Gamma \vdash \begin{pmatrix} \texttt{while}\ e : \\ S \end{pmatrix} : \Gamma_f}$$

Question #12
Let $\Gamma = [x \mapsto \{\mathbf{Z}\}]$ Type the following expressions in Γ: $x + 1$, $[\texttt{true}]$, $x + [3]$.

Solution:

$$\{\mathbf{Z}\} \qquad \{\mathbf{L}\} \qquad \{\mathbf{Z},\mathbf{L}\}$$

to be justified by rule applications, e.g. the last one is

$$\frac{\dfrac{\Gamma \vdash x : \Gamma(x)}{\Gamma \vdash x : \{\mathbf{Z}\}} \qquad \Gamma \vdash [3] : \{\{\mathbf{L}\}\}}{\Gamma \vdash x + [3] : \{\mathbf{Z},\mathbf{L}\}}$$

Question #13

Define the typing analysis for pop.

Solution:

$$\Gamma \vdash x, y := pop(e) : \Gamma[x \mapsto \{\mathbf{Z}\}, y \mapsto \{\mathbf{L}\}]$$

Question #14

In Figure 1c, what should be the type for z on the last line.

Solution: $\{\mathbf{Z};\mathbf{L}\}$

Question #15

Define the typing analysis for if. You may use a $\uplus$ operator such that:

$$(\Gamma \uplus \Gamma')(x) = \Gamma(x) \cup \Gamma'(x)$$

Solution:

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash S_1 : \Gamma_1 \qquad \Gamma \vdash S_2 : \Gamma_2}{\Gamma \vdash \begin{array}{l}\texttt{if } e : \\ \quad S_1 \\ \texttt{else} : \\ \quad S_2\end{array} : \Gamma_1 \uplus \Gamma_2}$$

Question #16

Explain the rule for while.

Solution: The difficulty for while is that variables may change type between loop executions. Therefor, the initial typing environment is not sufficient to obtain a complete description of the typing after the execution of the while.

Let us note Γ_f the final typing environment. It needs to be both bigger than the input typing environment ($\Gamma \subseteq \Gamma_f$), and than the typing environment after the while ($Gamma' \subseteq \Gamma_f$). We must then decide in which environment to type the while: it must be typed in both the initial one, and after any number of iterations. Γ_f satisfies both criterions, we thus state $\Gamma_f \vdash S : \Gamma'$.

Question #17 (Difficult)

Prove that the analysis always succeeds for expressions: for all e and Γ such that $Vars(e) \subset Dom(\Gamma)$, there exists a T such that $\Gamma \vdash e : T$. Proof is by induction, only consider the case for $+$ and $[e]$.

Solution:

- Cas +:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 + e_2 : T_1 \cup T_2}$$

  par HI: $Vars(e_1 + e_2) \subset Dom(\Gamma)$ Pour i=1,2, $Vars(e_i) \subset Vars(e_1 + e_2) \subset Dom(\Gamma)$. On a donc par induction sur les appels en e_1 et e_2, il existe T_1 et T_2 tq $\Gamma \vdash e_i : T_i$. $T_1 \cup T_2$ est bien défini.

- Cas singleton:

$$\frac{}{\Gamma \vdash [e] : \{\mathbf{L}\}}$$

  trivialement, on a $T = \{\mathbf{L}\}$.

3.2 Type-directed Compilation

We now have a more precise account of which types are admissible at each point in the program, thanks to the typing analysis developed in the previous section. For simplicity, we consider a function $Type : \mathcal{E} \to \mathcal{P}(\mathcal{T})$ that gives us the set of types inferred by the typing analysis for any expression of the program (see usage below).

Question #18

Consider Figure 1b. Consider that l is always a list, i.e. $Type(l) = \{\mathbf{L}\}$. Give an optimized version of the compilation of this program that uses the information provided by the typing analysis.

Solution:

```
if (l == []){b := false}
else {b := true}
if (b) then
  x, l := pop(l)
else
  x := 0
```

Or even:

```
if (l == []) then
  x, l := pop(l)
else
  x := 0
```

We specify the optimised compilation as an additional set of compilation rule that are redundant with the existing ones but specialised to optimisable cases. The idea is that we first try to apply optimised rules and fallback to the non-optimised compilation if no optimised rule applies. For example we add the following rule for if, when the condition is always a boolean:

$$\frac{Type(e) = \{\mathbf{B}\} \qquad [\![e]\!] = x_w, S_w \qquad [\![S_0]\!] = S_{w,0} \qquad [\![S_1]\!] = S_{w,1}}{\left[\!\left[\begin{array}{l}\texttt{if } e : \\ \quad S_0 \\ \texttt{else} : \\ \quad S_1\end{array}\right]\!\right] = S_w; \texttt{if } x_w \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}}$$

Question #19

Complete the following rule for if:

$$\frac{Type(e) = \{\mathbf{L}\} \qquad \text{TO COMPLETE}}{\left[\!\left[\begin{array}{l}\texttt{if } e : \\ \quad S_0 \\ \texttt{else} : \\ \quad S_1\end{array}\right]\!\right] = \text{TO COMPLETE}}$$

Solution:

$$\frac{Type(e) = \{\mathbf{L}\} \qquad [\![e]\!] = x_w, S_w \qquad [\![S_0]\!] = S_{w,0} \qquad [\![S_1]\!] = S_{w,1}}{\left[\!\left[\begin{array}{l}\texttt{if } e : \\ \quad S_0 \\ \texttt{else} : \\ \quad S_1\end{array}\right]\!\right] = S_w; \texttt{if } (x_w == \texttt{[]}) \texttt{ then } S_{w,0} \texttt{ else } S_{w,1}}$$

Question #20

Complete the following rule for +:

$$\frac{Type(e) = \{\mathbf{Z}, \mathbf{L}\} \qquad Type(e') = \{\mathbf{Z}, \mathbf{L}\} \qquad \text{TO COMPLETE}}{[\![e + e']\!] = \text{TO COMPLETE}}$$

Explain informally in two sentences how this rule would optimise the compilation of Figure 1c.

Solution:

$$\frac{Type(e) = \{\mathbf{Z}, \mathbf{L}\} \qquad Type(e') = \{\mathbf{Z}, \mathbf{L}\} \qquad [\![e]\!] = S_1, x_1 \qquad [\![e']\!] = S_2, x_2 \qquad x_f \text{ fresh}}{[\![e + e']\!] = \begin{array}{l} S_1; S_2; \\ \texttt{if } (\texttt{is\_int}(x_1) \,\&\&\, \texttt{is\_int}(x_2))\{x_f := x_1 + x_2\} \\ \texttt{else if } (\texttt{is\_list}(x_1) \,\&\&\, \texttt{is\_list}(x_2))\{x_f := concat(x_1, x_2)\} \\ \texttt{else } \{\texttt{Fail}\} \end{array}, x_f}$$

It avoids an extra test for Figure 1c in $x + y$, since x and y can't be booleans.

Question #21 (Difficult)

Bonus question. How would you define a type-based optimised compilation for while?

Solution: We can leverage the while typing rule to ensure collect the type of the conditional expression both in the initial test, and in the tests in between loops. We must gather all these types and introduce a specialize truthyness test that account for them all.