CS 3110 Problem Set 5 (Project Part I): Concurrent Language Interpreter

Due date: 11:59 PM, November 13, 2009

Updates

- Nov. 5: clarified what we are expecting for the proof of correctness of `bsearch`.

1 Introduction

In this assignment, you will finish the implementation of an interpreter for a concurrent functional language called CL. The purpose is for you to gain experience understanding, modifying, and optimizing a complex piece of code. Your goal is to make the program work correctly and efficiently, without changing it more than necessary. In addition to the implementation of CL, there are written problems relating to program verification.

A CL program has multiple, parallel threads of execution. Each thread can communicate with other threads through message passing. Threads can also start other threads to carry out tasks, possibly in cooperation with the original thread. CL programs can interact with an external environment that provides additional functionality, such as I/O. In the next assignment, you will use your interpreter to implement a game that uses robots controlled by a program written in CL, with each robot controlled by a different thread.

We have provided a partial implementation of the CL interpreter. The implementers were seemingly very lazy and didn't finish the implementation of all CL expressions. They also didn't think about how they could use data structures to accelerate the various operations performed by the interpreter. As a result, their interpreter is both broken and slow. You will fix this.

The missing piece of CL is the implementation of the `match` expression, which is similar to but different from the OCaml `match`. You will also speed up the interpreter by figuring out where time goes during execution, and choosing the right data structures. For this assignment, we expect you to implement your *own* data structures rather than using data structures from the OCaml library. Where appropriate, we expect you to add new modules to the existing program.

We are not expecting you to rewrite the interpreter. In fact, you will be most successful if you think carefully where the time is currently going, and solve the performance problems by changing existing code as little as possible, and making additions in a modular way. We expect you to use *profiling* to gain an understanding of the program performance and where your changes can have the biggest impact.

As always, your programs must compile without any warnings. Programs that do not compile or compile with warnings may receive an automatic zero. Files submitted should *not* have any lines longer than 80 characters, and ideally all lines should be less than 78 characters long. We will evaluate your problem set on several different criteria: the specifications you write, the correctness of your implementation, code style, efficiency, and your validation strategy. This is a complex

problem set, and you will be building on your PS5 solution for PS6, so we strongly recommend starting early.

2 The CL language

2.1 Overview

The CL language has some interesting features. It is a concurrent language in which multiple threads can execute simultaneously, and interact with an external environment. Unlike ML, CL is an *dynamically typed* language. There is no type checker to keep you from writing code that produces type errors. However, type errors are caught at run time, stopping the thread that encounters them.

Threads. A running thread can launch another thread using the expression `spawn` e. The expression e is the CL expression that the newly created thread will execute independently of its parent thread.

Any given thread is either ready to take an evaluation step, or blocked, waiting for something to happen. Threads can block waiting to receive a message from a mailbox, and when interacting with the external environment.

Threads interact with their external environment using the expression form `do` e. This expression is evaluated by sending the value of e to the external environment. What happens depends on the external environment that the CL program is interacting with; the behavior of the external environment is not specified by the CL language. Typically, different possible values of e are interpreted as requests to perform different actions.

In the external environment provided for PS5, the `do` e expression is used for I/O. For example, the expression `do` 0 causes the external environment to ask the user to input a number, which is returned as the result of the expression. In the next assignment, you will modify the implementation of the external environment to allow CL threads to implement robots that sense and interact with the world around them.

Message Passing. CL manages thread communication and synchronization using *message passing*. Threads communicate by sending and receiving messages to and from *mailboxes*. A mailbox can be created using the expression `mailbox` e. Mailboxes are *asynchronous*, which means that a thread sending a message does not wait for a thread to receive it before going on.

Arrays. Unlike in OCaml, the arrays in CL are immutable. Update to an array is nondestructive, producing a new array that differs only in the one index updated. Both updating and reading from an array are expected to be fairly efficient.

Any integer is a valid index into CL arrays, including negative integers and indices never before used. Arrays do not have a length, so it is not possible to have an out-of-range index. Any type of value, including other arrays, can be stored as array elements. The entries in an array need not be of the same type.

An array is really a (total) function from integers to CL values. We use mathematical notation to describe the functions corresponding to CL arrays. If a is an array, we write $a(n)$ to mean the value in a at index n. We write $\{n \mapsto 0\}_n$ to represent the array that maps all integers n to 0. We write $a[n \mapsto v]$ to represent the array that maps n to v, but that maps every other index $n' \neq n$ to $a(n')$. None of this notation is allowed in CL programs; we just use it to describe how CL arrays work.

Values. There are only four types of values in CL:

- Integer constants n
- Functions `fun` id`->`e.
- Arrays a. These are functions from integers to CL values. They can't be written directly in a CL program, but they can appear during evaluation using the substitution model.
- Mailboxes $mailbox$. These are identifiers that are global to all threads in the program. Like arrays, they can't be used in a CL program, but appear during evaluation.

2.2 Expressions

A CL program can consist of the following expressions:

n	An integer constant, as in OCaml. Examples: -3, 0, 2.
unop e	Returns *unop* applied to the result of evaluation of e. *unop* is one of following unary operators: $-$ (negates an integer), `not` (logical operator) and `rand` (returns a random number between 0 and $n-1$ where n is the value of e).
e_1 *binop* e_2	Applies binary operator *binop* to the results of evaluations of the two expressions. Both e_1 and e_2 must evaluate to an integer. *binop* is one of the following operators: $+, -, *, /, \texttt{mod}, <, >, \leq, \geq, =$. For the last five operators the result will be 1 if the OCaml result is true, and 0 otherwise.
e_1 ; e_2	A sequence of expressions. It is evaluated similarly to an ML sequence. First expression e_1 is evaluated, possibly causing side effects. After that the result of e_1 is thrown away and expression e_2 is evaluated.
`let` $id = e_1$ `in` e_2	Binds the result of evaluating e_1 to the identifier id and uses the binding to evaluate e_2. Identifiers start with a letter and consist of letters, underscores, and primes.
`fun` id`->`e	An anonymous function with argument id and body e. The body is not evaluated until an argument is supplied to the function.
id	Identifier. Must be contained inside a `let` or `fun` expression with the same identifier name. Otherwise, an unbound identifier error occurs when the identifier is used.
e_0 e_1	Function application. Evaluates expression e_0 to a function value `fun` id `->` e, evaluates expression e_1 to a value v_1, binds v_1 to the identifier id and uses the binding to evaluate e.

`rec` id `in` e Introduces a recursive term named id. id is in scope in e, and id is bound to e. This expression can be used to implement recursive functions, e.g. `let` $fact =$ `rec` f `in fun ->` `if` $n = 0$ `then` 1 `else` $n * f(n-1)$ `in` $fact(3)$

`if` e_0 `then` e_1 `else` e_2 Similar to the ML `if/then/else` expression except that the result of expression e_0 is tested to see if it is positive. Examples: `if 1 then 1 else 2` evaluates to 1; `if 4<3 then 1 else 2` evaluates to 2; `if -1 then 2 else 3` evaluates to 3.

`match` e `with`
$(p_{10}, \ldots, p_{1(n_1-1)})$ `->` e_1
| ...
| $(p_{m0}, \ldots, p_{m(n_m-1)})$ `->` e_m

Evaluates expression e to a value that must be an array a. Then it compares the array a against the patterns in the m arms following. It evaluates the first arm whose pattern matches the array.

The mechanics of pattern matching are a bit different from in OCaml. Each pattern in the list looks like an array constructor. The ith pattern has the form $(p_{i0}, \ldots, p_{i(n_i-1)})$, where n_i is the number of elements in the ith pattern. Each of the pattern components p_{ij} is either `val` id for some identifier id, or else it is an arbitrary expression e. If the component p_{ij} has the form `val` id, it causes the variable id to be bound to the corresponding value $a(j)$. By contrast, a component p_{ij} that is an expression e is evaluated to a value v, and the whole pattern matches only if $v = a(j)$ for all such components. Expressions in patterns are only evaluated when the arm containing the pattern is being considered for matching, and not before.

For example, the following code evaluates to 1:

```
let x = 3 in
  match (1,2,3,4) with
    (x, 2, x, 2+2) -> 0
  | (val one, 2, x) -> one
  | (1, 2, val y, 3+1) -> 2
```

The first pattern is not matched because `x`=3, which doesn't match the array element 1. The second pattern *is* matched, because the pattern only checks array indices 0, 1, and 2. The third pattern would match, but the second pattern wins because it is earlier. The expression `2+2` is evaluated, but the expression `4+5` is not, because its arm is never reached.

`array` e Evaluates e to a value v, then creates a new array a that contains v in every element; that is, $\{n \mapsto v\}_n$. The value of the expression is this array a.

$(e_0, e_1, \ldots, e_{n-1})$ Evaluates e_i to v_i, then creates a new array a that contains v_i at the i^{th} index for $i = 0, 1, \ldots, n-1$ and 0 at every other index. The value is this

array a.

String literals are syntactic sugar for arrays of integers, in which each index of the array gives the ASCII code for the corresponding character. For example, ``` ``hello'' ``` is sugar for $(104, 101, 108, 108, 111)$.

a An array, which is actually a total map from integers to values. This kind of expression never appears in CL source code, but can occur during evaluation, according to the semantics given in Section 2.4.

$e_1[e_2]$ Evaluates expression e_1 to an array a and e_2 to an integer n. Returns $a(n)$, the value in the array at index n.

$e_1[e_2 := e_3]$ Evaluates expression e_1 to an array a, and expression e_2 to an integer n, and e_3 to a value v_3. Returns a new array $a[n \mapsto v_3]$: just like a, except that the element at index n is changed to v_3. Arrays are immutable, so a remains unmodified. The index n can be any integer; both positive and negative array indices are allowed.

`mailbox` e The result of this expression is a newly created mailbox. However, this expression first evaluates e to a value v. When this mailbox is empty and the operation `recv` is used on it, it returns v.

`send` e_1 `<-` e_2 This expression first evaluates e_1 to a mailbox $mailbox$ and e_2 to v_2. The value v_2 is inserted in $mailbox$, and the expression returns $mailbox$.

`recv` e This expression first evaluates e to a mailbox $mailbox$. A value v from $mailbox$ is removed, and the expression returns v. If the mailbox does not contain any values, the expression returns the value that was provided while creating the mailbox.

`wait` e This expression evaluates e to a mailbox $mailbox$. If no values are in $mailbox$, the thread blocks until a value is sent to $mailbox$. The result of this expression is always $mailbox$, which makes it convenient to do `recv (wait m)`.

There is no guarantee that there will be a message available in the mailbox when `wait` unblocks, because another thread could have used `recv` to read the message first.

`do` e This allows a thread to interact with the external world. First, expression e is evaluated to a value v which is then sent to the external world. The thread blocks waiting for the external world to provide a result. The return result of this expression can be any CL expression (it is specified by the external world, and need not be a value). The list of requests currently recognized by the external world is given in section 2.5.

`spawn` e This creates a new thread that evaluates the expression e concurrently with the existing threads.

We have provided for you a representation for expressions as the type `Ast.exp` in the file `eval/ast.ml`. Note that these expressions are often contained within values of the type `Ast.line`. `Ast.line` is a type that simply pairs an expression with the filename and line number at which it appeared. The evaluator uses these line numbers and filenames to give error messages that tell you

exactly where errors occurred.

2.3 Evaluation

A thread is represented by a unique thread identifier pid and expression e. The current state of the CL interpreter is a queue of threads. The interpreter repeatedly performs the following operation: it takes the thread at the head of the queue, performs a single evaluation step on its expression, and places the modified thread at the end of the queue. It is important that threads execute one step at time. If the interpreter evaluated a program down to a value all at once, the system would not be concurrent because only the thread being evaluated would be able to run. Therefore, the interpreter evaluates threads in single steps. Given an expression, the evaluator finds the leftmost subexpression that can be reduced, and reduces this subexpression.

If a thread expression is not a value but there is no legal reduction to be performed, evaluation of that thread is *stuck*. This is a run-time error that terminates the thread. For example, an expression like `1/0` is stuck because it has a division by zero; the expression `5[0]` is stuck because there is a type error: `5` is not an array.

2.4 Reductions

The list of possible reductions that can be performed during evaluation is given below. These reductions are similar to the reductions you have learned for OCaml. Letters v stand for values, and letters e for expressions which may or may not be values. The notation $e{\cdot}\{v/x\}$ is an explicit substitution term (see Section 2.9).

There are several expressions in which reductions occur before the evaluation of their subexpressions. These expressions are the following: `let` $id = v$ `in` e, `if` v `then` e_1 `else` e_2, `fun` id `->` e, `rec` id `in` e, `match` v `with` $(p_{10}, id_1, \ldots, id_{n-1})$ `->` $e_1 \mid \ldots$, `spawn` e, and $v\ ;\ e$. The v's indicate subexpressions that must be fully evaluated before the expression can be reduced, and the e's indicate subexpressions that are not evaluated until after the reduction of the whole expression. In a `match` expression, the subexpressions of the first arm

$$
\begin{array}{rcll}
\textit{unop}\ v & \longrightarrow & v' & \text{where } v' = \textit{unop}\ v \text{ mathematically.} \\
v_1\ \textit{binop}\ v_2 & \longrightarrow & v' & \text{where } v' = v_1\ \textit{binop}\ v_2 \text{ in OCaml, for operators } \texttt{+}, \texttt{*}, \texttt{/}, \texttt{mod}. \text{ For operators } <, >, \leq, \geq, \text{ and } =, \text{ the result is 1 where it would be } \texttt{true} \text{ in OCaml, and 0 if it would be } \texttt{false}. \\
v;\ e & \longrightarrow & e & \\
\texttt{let}\ id = v\ \texttt{in}\ e & \longrightarrow & e{\cdot}\{v/id\} & \\
\texttt{rec}\ id\ \texttt{in}\ e & \longrightarrow & e{\cdot}\{\texttt{rec}\ id\ \texttt{in}\ e/id\} & \\
(\texttt{fun}\ id\ \texttt{->}\ e)\ v & \longrightarrow & e{\cdot}\{v/id\} & \\
\texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_1 & \text{where } n > 0 \\
\texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_2 & \text{where } n \leq 0 \\
\texttt{match}\ a\ \texttt{with}\ (p_{10}, p_{11}, \ldots, p_{1(n_1-1)})\ \texttt{->}\ e_1 \mid rest & \longrightarrow & e_1 \cdot \{v_j/id_j\} & \text{where the indices } j \text{ are those with the property that } p_j \text{ has the form } \texttt{val}\ id_j. \text{ The pattern must match: that is, for all } k \text{ where } p_k
\end{array}
$$

has the form v_k instead, $a(k) = v_k$. The metavariable $rest$ stands for the rest of the arms of the `match`, if any.

$\texttt{match}\ a\ \texttt{with}\ (p_{10}, p_{11}, \ldots, p_{1(n_1-1)})\ \texttt{->}\ e_1 \mid rest \longrightarrow \texttt{match}\ a\ \texttt{with}\ rest$

where the pattern does not match: for some k where p_k has the form v_k, $a(k) \neq v_k$. It is a run-time error if the pattern does not match and $rest$ is empty.

The rules for the array operations are as follows:

$a\ [v_1] \longrightarrow v_2$

where a is an array. The value v_2 is $a(v_1)$, the value at the v_1^{th} index of a.

$\texttt{array}\ v \longrightarrow a$

where a is a new array that contains v in every element. That is, $a = \{n \mapsto v\}_n$

$(v_0, v_1, \ldots, v_{n-1}) \longrightarrow a$

where a is a new array with its contents initialized to an array that contains v_i at the i^{th} index for $i = 0, 1, \ldots, n-1$ and 0 at every other index. That is, $a = \{n \mapsto 0\}_n[0 \mapsto v_0, \ldots, n-1 \mapsto v_{n-1}]$

$a[v_1 := v_2] \longrightarrow a'$

where a is an array. The return value is a new array a' which is a with the v_1^{th} index containing the value v_2

Finally, the reductions for concurrent constructs are:

$\texttt{mailbox}\ v \longrightarrow mailbox$

The result is a new mailbox $mailbox$ that returns v on a `recv` operation when the mailbox is empty.

$\texttt{send}\ mailbox\ \texttt{<-}\ v \longrightarrow mailbox$

Effects: value v is inserted in the mailbox $mailbox$. The result is $mailbox$.

$\texttt{recv}\ mailbox \longrightarrow v$

Retrieves a value v from the mailbox $mailbox$. Two messages sent by the same thread will be received in the order in which they were sent. No other guarantees are placed on the order of delivery. If the mailbox is empty, the value that was provided while creating the mailbox is returned. Effects: the value v is removed from the mailbox.

$\texttt{wait}\ mailbox \longrightarrow mailbox$

Blocks the current thread until a message has been sent to the mailbox $mailbox$.

`do` $v \longrightarrow e$

where e is the expression returned by the external world
Effects: send `doAction(`pid`,` v`)` to the external world where pid is the thread identifier. The external world provides the expression e.

`spawn` $e \longrightarrow n$

Effects: ask the external world for a fresh thread identifier pid'. If this succeeds, launch a new thread with the identifier pid' expression e, and a copy of the environment of the current thread; the result is 1. If the world does not permit a new thread to be spawned, the result is 0

Notice that because expressions may have side effects, it is critical that expressions are evaluated left to right. For example, e_1 *binop* e_2 must be evaluated as

$$e_1 \textit{ binop } e_2 \longrightarrow v_1 \textit{ binop } e_2 \longrightarrow v_1 \textit{ binop } v_2 \longrightarrow v$$

2.5 The external environment

Currently the `do` action performs simple I/O operations, though in PS6 it will be a general mechanism for interacting with the world. The following actions are currently provided:

- `do 0` : read a number from the input and return it to the interpreter
- `do (1,` v`)` : print v and returns v. Any value may be printed. Functions will result in printing ⟨`function`⟩, and mailboxes will print ⟨`mailbox`⟩.
- `do (2, a)` : print the array a as a string. Returns 1 if the array obeys the string rep invariant (i.e., length at index -1). Thus, `do (2, "hi")` will print out the string "hi".
- `do 4` : reads a string from the input, and returns it to the interpreter as an array.

2.6 Configurations

A *configuration* is the state of the entire interpreter at a particular point during execution. The configuration consists of a set of threads, each of which has a currently executing expression and a thread id.

We can describe a single thread as a tuple $\langle pid, e \rangle$. The entire interpreter configuration is the current queue of threads:

$$\langle \langle pid_1, e_1 \rangle, \ldots, \langle pid_n, e_n \rangle \rangle$$

The thread at the head of the queue, thread 1, is the one that will take the next evaluation step and be pushed to the end of the queue. Suppose that this thread takes the evaluation step $e_1 \longrightarrow e_1'$. Then the effect of this step on the configuration as a whole is this:

$$\langle\langle pid_1, e_1\rangle, \langle pid_2, e_2\rangle, \ldots, \langle pid_n, e_n\rangle\rangle \longrightarrow \langle\langle pid_2, e_2\rangle, \ldots, \langle pid_n, e_n\rangle, \langle pid_1, e_1'\rangle\rangle$$

The type for configurations, `Configuration.configuration`, is defined in the source file `eval/configuration.ml`. A single step of the interpreter is performed by the function `Evaluation.stepConfig` in `eval/evaluation.ml`.

2.7 Creating threads

Threads can create other threads by calling `spawn` e. As a result, a new thread will be added to the list of threads. The two threads are able to communicate with each other through mailboxes created before evaluating `spawn`.

2.8 Errors and termination

If a thread has evaluated to a value, it *terminates* and is deleted from the list of threads. Thus, we have the following evaluation rule:

$$\begin{array}{rl} & \langle\langle pid_1, v_1\rangle, \langle pid_2, e_2\rangle, \ldots, \langle pid_n, e_n\rangle\rangle \\ \longrightarrow & \langle\langle pid_2, e_2\rangle, \ldots, \langle pid_n, e_n\rangle\rangle \end{array}$$

In incorrect programs, expressions can encounter run-time errors, such as run-time type errors. Run-time type errors are expressions that are not value but for which there is no legal reduction. If a thread in a CL program encounters a run-time error, that single thread immediately terminates. Other threads are not directly affected, however. Errors should terminate the thread encountering them, but do not affect other running threads.

2.9 Substitutions

To speed up evaluation, the interpreter does not eagerly substitute for all unbound occurrences when a variable is bound in a function call or a `let`. Instead, the interpreter uses an *explicit* substitution model, in which the substitution $e\{v/x\}$ is represented by a explicit substitution term written here as $e{\cdot}\{v/x\}$. For example, $2{\cdot}\{\}$ (that is, 2 with an empty substitution) is equivalent to $x{\cdot}\{2/x\}$; they both evaluate in a single step to 2. During evaluation, substitutions are delayed till variables need to be evaluated. This means the interpreter avoids doing substitution work that is not needed.

The substitution rules are given below. The notation $\{\vec{v}/\vec{x}\}$ is shorthand for a substitution for multiple variables x_i at once: $\{v_1/x_1, \ldots, v_n/x_n\}$. The notation $\{\vec{v}/\vec{x}\} - x$ represents set $\{\vec{v}/\vec{x}\}$ with the binding for x (if any) removed, and $\{\vec{v}/\vec{x}\} + \{\vec{v}'/\vec{x}'\}$ represents the union of substitutions in $\{\vec{v}/\vec{x}\}$ and $\{\vec{v}'/\vec{x}'\}$, except that $\{\vec{v}'/\vec{x}'\}$ overrides $\{\vec{v}/\vec{x}\}$ on any variable that both substitute for.

$$\begin{array}{rcl} x_i \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & v_i \\ (\textit{unop}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \textit{unop}\ (e \cdot \{\vec{v}/\vec{x}\}) \\ (e_1 \textit{binop}\ e_2) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & (e_1 \cdot \{\vec{v}/\vec{x}\})\ \textit{binop}\ (e_2 \cdot \{\vec{v}/\vec{x}\}) \end{array}$$

$$
\begin{array}{rcl}
(e_1;\ e_2)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & (e_1\cdot\{\vec{v}/\vec{x}\});\ (e_2\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{let}\ id = e_1\ \texttt{in}\ e_2)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{let}\ id = (e_1\cdot\{\vec{v}/\vec{x}\})\ \texttt{in}\ (e_2\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{rec}\ id\ \texttt{in}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & e\cdot\{\vec{v}/\vec{x}\} + \{\texttt{rec}\ id\ \texttt{in}\ e/id\} \\
(\texttt{fun}\ id\ \texttt{->}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{fun}\ id\ \texttt{->}\ (e\cdot\{\vec{v}/\vec{x}\} - id) \\
(e_1\ e_2)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & (e_1\cdot\{\vec{v}/\vec{x}\})\ (e_2\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{if}\ (e_1\cdot\{\vec{v}/\vec{x}\})\ \texttt{then}\ (e_2\cdot\{\vec{v}/\vec{x}\})\ \texttt{else}\ (e_3\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{match}\ e_1\ \texttt{with}\ p\texttt{->}\ e_2\ \ldots)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{match}\ (e_1\cdot\{\vec{v}/\vec{x}\})\ \texttt{with}\ p\cdot\{\vec{v}/\vec{x}\}\texttt{->}\ (e_2\cdot\{\vec{v}/\vec{x}\})\ \ldots \\
(e_0, e_1, \ldots, e_{n-1})\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & ((e_0\cdot\{\vec{v}/\vec{x}\}), (e_1\cdot\{\vec{v}/\vec{x}\}), \ldots, (e_{n-1}\cdot\{\vec{v}/\vec{x}\})) \\
(\texttt{array}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{array}\ (e\cdot\{\vec{v}/\vec{x}\}) \\
(e_1[e_2])\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & (e_1\cdot\{\vec{v}/\vec{x}\})[(e_2\cdot\{\vec{v}/\vec{x}\})] \\
(e_1[e_2 := e_3])\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & (e_1\cdot\{\vec{v}/\vec{x}\})[(e_2\cdot\{\vec{v}/\vec{x}\}) := (e_3\cdot\{\vec{v}/\vec{x}\})] \\
(\texttt{mailbox}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{mailbox}\ (e\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{send}\ e_1\ \texttt{<-}\ e_2)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{send}\ (e_1\cdot\{\vec{v}/\vec{x}\})\ \texttt{<-}\ (e_2\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{recv}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{recv}\ (e\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{wait}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{wait}\ (e\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{do}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{do}\ (e\cdot\{\vec{v}/\vec{x}\}) \\
(\texttt{spawn}\ e)\cdot\{\vec{v}/\vec{x}\} & \longrightarrow & \texttt{spawn}\ (e\cdot\{\vec{v}/\vec{x}\}) \\
(e\cdot\{\vec{v}/\vec{x}\})\cdot\{\vec{v'}/\vec{x'}\} & \longrightarrow & e\cdot(\{\vec{v}/\vec{x}\} + \{\vec{v'}/\vec{x'}\})
\end{array}
$$

The following example illustrates evaluation steps using explicit substitution:

$$
\begin{array}{l}
\texttt{let}\ x = 1\ \texttt{in}\ \texttt{let}\ f = \texttt{fun}\ z\ \texttt{->}\ x + z\ \texttt{in}\ f\ 3 \\
\longrightarrow\ (\texttt{let}\ f = \texttt{fun}\ z\ \texttt{->}\ x + z\ \texttt{in}\ f\ 3)\cdot\{1/x\} \\
\longrightarrow\ \texttt{let}\ f = (\texttt{fun}\ z\ \texttt{->}\ x + z)\cdot\{1/x\}\ \texttt{in}\ (f\ 3)\cdot\{1/x\} \\
\longrightarrow\ \texttt{let}\ f = (\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})\ \texttt{in}\ (f\ 3)\cdot\{1/x\} \\
\longrightarrow\ (f\ 3)\cdot\{1/x\}\cdot\{(\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})/f\} \\
\longrightarrow\ (f\ 3)\cdot\{1/x, (\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})/f\} \\
\longrightarrow\ f\cdot\{1/x, (\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})/f\}3\cdot\{1/x, (\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})/f\} \\
\longrightarrow\ (\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})\ 3\cdot\{1/x, (\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})/f\} \\
\longrightarrow\ (\texttt{fun}\ z\ \texttt{->}\ (x + z)\cdot\{1/x\})\ 3 \\
\longrightarrow\ (x + z)\cdot\{1/x\}\cdot\{z/3\}\ \longrightarrow\ (x + z)\cdot\{1/x, 3/z\} \\
\longrightarrow\ x\cdot\{1/x, 3/z\} + z\cdot\{1/x, 3/z\}\ \longrightarrow\ 1 + z\cdot\{1/x, 3/z\} \\
\longrightarrow\ 1 + 3\ \longrightarrow\ 4
\end{array}
$$

3 Using the interpreter

3.1 File structure

The interpreter code is structured as follows:

- `ast.ml`: definitions of basic types (`AST.exp`)
- `config.ml`: definition of the configuration type
- `evaluation.ml`: performs a single step of the main interpreter loop. The evaluation searches for the leftmost subexpression to reduce, then calls the reduction function.

- `concurrency.ml`: defines all the mailbox operations
- `arrayCL.ml`: defines the type of arrays
- `substMap.ml` defines the type of the substitution map
- `reduction.ml`: defines the one-step reduction function.
- `world.ml`: interface for interaction with the external world
- `debug.ml`: interface for debugging
- `cl/*.cl`, a few sample CL programs

3.2 Running CL code

You will build a custom toplevel containing all of your PS5 code. We have provided a script called `buildToplevel.bat` (which can be run on both Windows and Unix) which creates such a toplevel. To start the toplevel, execute the script `runToplevel.bat`. From the toplevel, you can enter the debugging mode using the command:

`Debug.debug` "a string representing an CL program"

You will see a prompt (>). You can get the list of available commands by typing "help". These are some commands for quick start:

- `s`: steps one step and shows the new stepped expression
- `r`: runs until the end
- `load` *filename*: resets the interpreter and loads a file with an CL program
- `h`: gives you the help message and shows you many more commands
- `q`: quits the debugger

There are many other helpful functions and debugger commands; see `debug.ml` for more details. If you feel that the debugging tools implemented are inadequate, feel free to modify them.

4 Your task

Part 1: Match Evaluation

Finish the implementation of the `match` expression. You will have to make changes to the following files:

- `eval/evaluation.ml`
- `eval/reduction.ml`

Part 2: Performance Improvements

The current CL interpreter is very inefficient. It is your task to discover where these bottlenecks lie and to improve the performance of the interpreter. You should be able to speed up the interpreter by an order of magnitude if you do your job right. You should also document any changes you made and explain how and why they improve performance. You are allowed to modify any code in the `eval` directory. However, you should not modify the `evaluation.mli` file. Gratuitous changes may result in a penalty.

As on PS3, the group that produces the fastest correct interpreter implementation will receive a bonus.

You will want to think about the right data structures to implement performance-critical abstractions.You may use any data structures you find in the OCaml Core Library (e.g., arrays and lists). However, you are *not* allowed to use the additional data structures found in the OCaml Standard Library except List and Array, though you may implement your own versions of any data structures if you want to.

To help you figure out where your time is going in the interpreter, you will probably find the OCaml profiler to be helpful. A profiler is a tool that records where time is being spent in your program. It can then generate various useful reports that will help you identify performance bottlenecks. We will expect you to show us before-and-after profiles for your interpreter running a standard benchmark, and to explain how these profiles show that you did your job well.

Unfortunately, Windows does not support profiling OCaml programs very well. For this reason, you will need to profile your code on Linux. We will host a demo session where we will show you how to use the OCaml profiler and Linux. If you have a CSUGLab account, you can access computers running Linux (`csug11–csug15` or `linus`, `schroeder`, `marcie`, `redbaron`, `sally`) by following the instructions at:
`http://www.csuglab.cornell.edu/Info/linux-info.html`. If you do not have a CSUGLab account, you can obtain one:
`http://www.csuglab.cornell.edu/Info/accounts.html` The course staff can help you find a Linux machine and profile your code if you go to consulting hours or office hours.

Running the profiler

Once you've logged into Linux, make sure that $ocaml$ and $gprof$ are installed (they should already be installed on the CSUGLab machines).

1. From the command line, navigate to the directory that contains your PS5 solution.

2. Run the `buildExecutable.sh` script by entering the command `./buildExecutable.sh`. It should create an executable called `runcl` in your current directory. `runcl` is a program that takes the name of a CLfile as an argument on the command line and executes it. The `buildExecutable.sh` script builds `runcl` with extra profiling code. This extra code records how much time is being spent in each part of your OCaml code each time you execute `runcl`.

   If you get an error saying you do not have permission to run the script, you will need to change its permissions with the command `chmod +x buildExecutable.sh`.

3. Now choose a CL program (for example, `cl/fibo.cl`) that you want your interpreter to evaluate, and use the command `./runcl cl/fibo.cl` to evaluate it. `runcl` will generate profiling information while it runs.

4. To view the profiling information that you just recorded, you will use a program called `gprof`. You can do this with the command `gprof runcl`. This command will print a lot of text to the console, so you may prefer to redirect its output to a file:
   `gprof runcl > results.txt`

   One particularly useful table is the "flat profile". It lists each function that was called along with the percentage of the total time that was spent in it, as in the following:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.00      0.10     0.10  1000012     0.00     0.00  camlEvaluation__stepProcess_116
 11.25      0.15     0.05  1700014     0.00     0.00  camlReduction__subst_119
 10.00      0.19     0.04  4500033     0.00     0.00  camlReduction__isValue_103
  7.50      0.22     0.03  1000018     0.00     0.00  compare_val
  7.50      0.25     0.03        1    30.00   350.01  camlDebug__step_69
  5.00      0.27     0.02  1000012     0.00     0.00  camlEvaluation__stepAndUpdate_251
  5.00      0.29     0.02   900015     0.00     0.00  caml_compare
  3.75      0.30     0.02   400004     0.00     0.00  camlReduction__reduce_226
  2.50      0.31     0.01  1800072     0.00     0.00  caml_string_length
  2.50      0.32     0.01  1000012     0.00     0.00  camlEvaluation__stepConfig_260
  2.50      0.33     0.01   900006     0.00     0.00  camlSubstMap__compare_89
  2.50      0.34     0.01   200001     0.00     0.00  camlReduction__funOfBinop_187
```

   The "cumulative seconds" is just a running total of the "self seconds" column. The "self seconds" column measures the time spent in each function, not including any functions it calls.

   Later in the `gprof` output you will find breakdowns of where time was spent within each function, including functions they call. For example, if you wanted to see whether functions called by `Evaluation.stepProcess` were spending a lot of time, that's the place to look.

Part 3: Source Control

You are required to use a source control system like CVS or SVN. Submit the log file that describes your activity. If you are using CVS, this can be obtained with the command `cvs log`. CVS is supported in the CSUG Lab. For information on how to get started with CVS there, read the CVS/CSUG handout

Part 4: CL Implementation

CL doesn't have as many built-in operations and data structures as OCaml. Fortunately, many things we're used to having in other languages are easy to implement, using higher-order functions and other CL features.

1. Using CL, implement a function `foldi` that folds over a range of integers, much like `fold_left` folds over elements of a list. That is, `foldi body init start end` should apply the function `body` to every integer between `start` and `end`. The function `body` has two arguments; the first is the accumulated value (which starts at `init`) and the second is the current integer. For example,

   ```
   foldi (fun acc -> fun i -> do (1, i); acc+i) 0 1 5
   ```

   should print the numbers from 1 to 5, and its value should be 15.

   Your implementation should include a test harness for `foldi`.

2. A priority queue is a queue that allows elements to be pushed (enqueued) in any order, but when elements are dequeued, they come out in order of their priority.

   Write a test program that implements a concurrent, thread-safe shared priority queue in CL. That is, multiple threads should be able to use your priority queue concurrently. An element enqueued onto the priority queue by one thread can be dequeued by a different thread. In this context, thread-safe means that operations of the priority queue should not interfere with each other. If two threads attempt to dequeue simultaneously, they should never get the same object. Nor should objects get lost from the queue if there are concurrent enqueues or dequeues. Hint: use message passing.

   Be sure to write specs for any priority queue operations you define. Your implementation does not have to be as efficient as possible, but we will give bonus points for especially efficient implementations. Your program should include a test harness that creates two threads which both enqueue and dequeue 1000 elements.

Part 5: Written problems

There are three written parts to this assignment.

(a) Consider the problem of multiplying two n-bit integers. One of the more efficient methods for performing this task is the *Karatsuba* algorithm, given below:

```
Requires: x, y both n bit numbers
Returns: x × y
let rec karatsuba(x, y, n) =
 if (n = 1) then x*y
 else
   let x1 = first n/2 bits of x in
   let x2 = last n/2 bits of x in
   let y1 = first n/2 bits of y in
   let y2 = last n/2 bits of y in
   let U = karatsuba(x1, y1, n/2) in
   let V = karatsuba(x2, y2, n/2) in
   let W = karatsuba(x1 - x2, y1 - y2, n/2) in
   let Z = U + V - W in
   (2 ^ n) * U + 2 ^ (n/2) * Z + V
```

Derive a recurrence relation for Karatsuba's algorithm, explaining each term. Then solve for the closed form and prove your answer by induction. For simplicity, assume that $n = 2^k$ for some k. Additionally, any addition operations are performed in $O(n)$ time, as well as any multiplications by 2^k, since this can be accomplished by left-shifting the bits in the multiplicand.

How does this compare to multiplication as normally done by hand?

(b) Consider the function `bsearch`:

```
(* Requires: ∃ x such that a ≤ x, x ≤ b, f(x) = 0,
 *           and ∀ y,z (y ≤ z ⇒ f(y) ≤ f(z))
 * Returns: r = bsearch f a b, where a ≤ r, r ≤ b, and f(r) = 0
 *)
let rec bsearch f a b =
    if a = b then a
    else let m = (a+b)/2 in
    if f(m) < 0 then bsearch f m b
    else bsearch f a m
```

(a) This function contains an error. Identify it and explain how to fix it.

(b) Prove partial correctness of the corrected function using Hoare logic. Identify the step of the proof that would fail with the uncorrected version and give a brief explanation why. You may use reasonable axioms for algebraic manipulation of expressions over integers.

Your proof probably won't be small enough to write as one proof tree. Feel free to break it down into smaller trees that can be assembled into a single proof. Just make it clear how that assembly is done. (This amounts to proving lemmas.)

The Hoare logic proof will have several predicate logic premises. You don't have to give a formal proof of these premises, though you may. It is enough to give an informal proof: that is, a convincing English-language argument that each premise is true.

(c) Consider the function `filter`:

```
(* Requires: True
 * Returns: elements(filter f l) = { e ∈ elements(l) | f e }
 *)
let filter f l =
  let rec filter_t f l acc =
    match l with
      [] -> acc
    | h::t -> if f h then filter_t f t h::acc else filter_t f t acc
  in
    filter_t f l []
```

Write a clear specification for `filter_t` that would enable this code to be proved correct. That is,

(a) Its precondition should be satisfied by its use in `filter`.

(b) Its postcondition should satisfy the postcondition of `filter`.

(c) The precondition of its recursive call should be satisfied assuming its precondition is satisfied.

(d) The postcondition of its call should be satisfied assuming the postcondition of its recursive call is satisfied.

Argue *informally* that each of these four criteria are satisfied by your spec for `filter`.

Files to submit

- `PS5.zip`: A zip file containing all the files to run the interpreter
- `foldi.cl`: Foldi implementation file
- `priority.cl`: CL Priority Queue implementation file
- `written.txt` or `written.pdf`: Written problems solution file
- `ps5.log`: your CVS logs
- `design_overview.txt` or `design_overview.pdf`: An overview document for your assignment, as in the previous two assignments. Be sure to describe the changes you made and to explain how and why they improved performance.