CS 312 Problem Set 5 (Project Part I): Concurrent Language Interpreter

Due date: 11:59 PM, April 10, 2008

---

1 Introduction

In this assignment, you will finish the implementation of an interpreter for a concurrent functional language called CL. In addition to the implementation of CL, there are written problems.

A CL program has multiple, parallel threads of execution. CL also has some imperative features. Each thread can communicate with other threads through a global shared memory. 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 **typecase** expression. You will also speed up the interpreter by figuring out where time goes during execution, and choosing data structures (and possibly designing new modules) appropriately. We are not looking for you to rewrite the interpreter. In fact, you will be most successful if you figure out carefully where the time is currently going, and solve the performance problems by changing existing code as little as possible, adding new code in as modular a way as possible.

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. Get your design right from the beginning and the rest will go more smoothly.

Summary of changes and clarifications

- March 30: requirements for arrays and priority queues clarified.
- March 31: due date extended to April 10
- April 6: clarifications provided for the use of any label in the typecase evaluation and part (a) of the written problem.

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. It has arrays, which can be updated imperatively, and threads can interact with an external environment. Unlike ML, CL is an *dynamically typed* language, so there is no type checker to keep you from writing code that produces type errors.

Threads. A running thread can launch another thread using the expression $\mathbf{spawn}\ e$. 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 acquire a lock, waiting for condition variables, and when interacting with the external environment.

Threads interact with their external environment using the expression form $\mathbf{do}\ e$. 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 $\mathbf{do}\ e$ expression is used for I/O. For example, the expression $\mathbf{do}\ 0$ causes the external environement 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.

Arrays. Each thread has access to a *global memory* that is shared by all threads. The global memory stores mutable arrays that can be updated imperatively. It does this by serving as a mapping from *locations* to arrays. Threads can communicate with each other by modifying these arrays.

Arrays in CL can accessed at both negative and positive indices; in fact, any integer is a valid index. Arrays do not have a length, so it is not possible to have an out-of-range index. Any type of value, including array locations, can be stored as array elements. The entries in the an array need not be of the same type. Accesses to array elements at any index should take constant time, even if they are accesses to indices outside the bounds of previously used indices.

Programmer implementing more advanced data structures in CL will find arrays an essential tool.

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

- Integer constants n
- Functions $\mathbf{fn}\ id$ => e
- Array locations *loc*.

CL has a limited form of pattern matching, through the $\mathbf{typecase}$ expression, which allows checking which of these three types an expression evaluates to. It also can bind array elements to variables.

Locks and condition variables. CL supports two mechanisms that allow threads to synchronize their activities. *Locks* can be used to permit at most one thread to access a given resource at a time. Once a thread *acquires* a lock, any other thread that tries to acquire it will block until the first thread *releases* it. *Condition variables* allow threads to wait until other threads signal them to continue. Both locks and condition variables are named using memory locations. It is important to realize, locations are just names, and there is *no direct connection* between the array at location *loc*, the lock that is named *loc*, or the condition variable that is named *loc*. They are three separate entities that share a name. In particular, acquiring a lock on a memory location *loc* does not mean that the array at location *loc* is automatically protected from access by other threads. It only has that effect if those other threads try to acquire the same lock.

Garbage collection. Garbage is data in global memory that will never be used again. Garbage collectors clean up garbage by finding memory locations that are not reachable, by following every chain of location references from a running thread. Any unreachable location can never be used again because there is no way to reach it. Unreachable locations should be periodically reclaimed and used for subsequent allocation requests. The signature file $\mathbf{gc.sig}$ describes an automatic garbage collector for the CL language. Occasionally the garbage collector is used to reclaim unused memory. In our CL interpreter, garbage collection is implemented using the simple *mark-and-sweep* algorithm.

2.2 Expressions

A CL program can consist of the following expressions:

n	An integer constant, as in SML. Examples: ∼3, 0, 2.
unop e	Returns *unop* applied to the result of evaluation of e. *unop* is one of following unary operators: $\sim$ (negates an integer), and $rand$ (returns a random number between 1 and n where n is the result of evaluation 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: $+, -, *, /, \mathbf{mod}, <, =$. For the last two operators the result will be 1 if the comparison 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.
$\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2$	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.

fn 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 **fn** expression with the same identifier name. Otherwise, an unbound identifier error will occur.
$e_0\ e_1$	Function application. Evaluates expression e_0 to a function value **fn** 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 fn** => **if** $n = 0$ **then** 1 **else** $n * f(n-1)$ **in** $fact(3)$
if e **then** e_1 **else** e_2	Similar to the ML **if/then/else** expression except that the result of expression e is tested for being greater than 0 (there are no booleans in CL). Examples: **if 1 then 1 else 2** returns **1**, **if 4<3 then 1 else 2** returns **2**.
while e_1 **then** e_2	This expression works similarly to an SML while loop. At each iteration of the loop, the condition expression e_1 is evaluated. If the result is greater than 0, e_2 is evaluated and then another iteration of the loop is executed. If the result is equal to or less than 0, 0 is returned.
typecase e **of** $(id_0, \ldots, id_{n-1})$ => e_1 $\mid$ **int** id => e_2 $\mid$ **fun** id => e_4 $\mid$ **any** id => e_5	Evaluates expression e to a value. If the result is a location it binds the elements of the array from indices 0 to $n-1$ to the corresponding variables $id_0, \ldots, id_n$. Otherwise, it binds the the result to **id** in the appropriate case. The result is the evaluation of the expression e_i of the matched case. Each of the cases is optional and can occur at most once, in any order. The case for **any** matches any value, and is allowed only if at least two of the other cases are missing. As in ML, all cases must be covered. The expression **typecase** e_1 **of any** id => e_2 is equivalent to **let** $id = e_1$ **in** e_2.
array e	Evaluates e to a value v, then creates a new memory location *loc* mapping to a new array that contains v in every element. Returns *loc*.
$(e_0, e_1, \ldots, e_{n-1})$	Evaluates e_i to v_i, then creates a new memory location *loc* mapping to a new array that contains v_i at the i^{th} index for $i = 0, 1, \ldots, n-1$ and 0 at every other index. Returns *loc*. 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 $(104, 101, 108, 108, 111)$.
loc	A memory location whose name is *loc*. This can be thought of as the address of the location. A location can only be generated using array constructors. The *loc* expression never appears in the source code of an CL program, but can occur during its evaluation, according to the semantics of CL given in Section 2.5.
$e_1[e_2]$	Evaluates expression e_1 to a location *loc* and e_2 to an integer n. Returns

the value stored in the array at index n. This operation is *not* affected by whether *loc* is currently locked. Locking a location only affects other attempts to lock the same location.

$e_1[e_2] := e_3$ Evaluates expression e_1 to a location *loc* and expression e_2 to a value v_2 and e_3 to a value v_3. Then it assigns the value v_3 to the v_2^{th} index of array stored at location *loc*. The return result of this expression is v_3. v_2 can be any interger since both positive and negative array indices are allowed. Note that this operation is *not* affected by whether *loc* is currently locked. Locking a location only affects other attempts to lock the same location.

$\mathbf{lock}\ e_1\ \mathbf{in}\ e_2$ This expression first evaluates e_1 to a location *loc*. The expression then reduces to **locked** *loc* e_2. If *loc* is not already locked, then the current thread acquires a lock for *loc*. If any other thread already has the lock, the thread will block waiting for the lock to be released. This is similar to the **synchronize** statement in Java.

$\mathbf{locked}\ loc\ \mathbf{in}\ e$ This term occurs during the evaluation of a **lock** expression, after the lock for *loc* has been acquired. Like *loc*, this expression never appears in CL source code, but can occur during evaluation once a lock has been acquired on *loc*.

The subterm e_2 is evaluated, while holding the lock. The result of the expression is the result of evaluating e_2. When evaluation of e_2 finishes, the lock is released; if any other threads are blocked waiting for this lock, one of them acquires the lock and is allowed to keep executing.

$\mathbf{wait}\ e_1\ \mathbf{until}\ e_2$ This expression first evaluates e_1 and e_2 to locations *loc*$_1$ and *loc*$_2$. It is a run-time error if the lock named by *loc*$_1$ is not held by the current thread. Atomically, the lock *loc*$_1$ is released and the current thread blocks. (Atomically, meaning that no other thread does anything in between those two actions.) The thread is blocked until some other thread signals this thread using the expression **signal** *loc*$_2$ (on the same location). Once signaled, the current thread starts running and tries to reacquire the lock *loc*$_1$ again before completing the evaluation of **wait** (Thus, it may block again if some other thread acquires the same lock.) The result of **wait** is always 0. This expression is similar to the **Object.wait()** method in Java.

$\mathbf{signal}\ e$ This expression first evaluates e to a location *loc*. The thread signals one of the threads waiting on *loc* (if any). No lock need be held. The result of **signal** is always 0. This is similar to the **Object.signal()** method in Java.

$\mathbf{do}\ e$ 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 2.6.

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 **ast/ast.sml**.

2.3 Precedence

The grammar of CL is similar to that of SML. There are a couple of things to watch out for. In particular, the sequential composition operator (;) has higher precedence than **let**, **case**, **fn**, **rec**, and **while**, all of which can end in an expression that uses ";". So the following two code samples mean the same thing:

let x $=$ 2 **in**
do x;
fn z **=>** **do** y; $x+1$

let x $=$ 2 **in** (**do** x; (**fn** z **=>** **do** y; $x+1$))

This gives CL code a somewhat different flavor than SML. If you're not sure about precedence, parentheses can always be used to make it explicit.

2.4 Evaluation

A thread is represented by a unique thread identifier pid and expression e. A current state of the interpreter is described by a queue of threads, as well as a global memory M and a map W of locations to waiting 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 (possibly modifying global memory), 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 must evaluate in steps. Given an expression, the evaluator finds the leftmost subexpression that can be reduced, and reduces this subexpression.

Note that reductions can occur on several expressions before evaluating all of their subexpressions. These expressions are the following: **let** $id = v$ **in** e, **if** v **then** e_1 **else** e_2, **while** v **then** e, **fn** id **=>** e, **rec** id **in** e , **typecase** v **of** $(id_0, id_1, \ldots, id_{n-1})$ **=>** $e_1 \mid \ldots$, **spawn** e, **lock** v **in** 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.

2.5 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 SML. 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 2.10).

$$\textit{unop}\ v \longrightarrow v' \qquad \text{where } v' = \textit{unop}\ v$$

$$\begin{array}{rcl}
v;\ e & \longrightarrow & e \\
\textbf{let}\ id = v\ \textbf{in}\ e & \longrightarrow & e{\cdot}\{v/id\} \\
\textbf{rec}\ id\ \textbf{in}\ e & \longrightarrow & e{\cdot}\{\textbf{rec}\ id\ \textbf{in}\ e/id\} \\
(\textbf{fn}\ id\ \texttt{=>}\ e)\ v & \longrightarrow & e{\cdot}\{v/id\} \\
\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \longrightarrow & e_1 \qquad\qquad v \in \{1,2,3\ldots\} \\
\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \longrightarrow & e_2 \qquad\qquad \text{all other } v \\
\textbf{while}\ e_1\ \textbf{then}\ e_2 & \longrightarrow & \textbf{if}\ e_1\ \textbf{then}\ e_2;\textbf{while}\ e_1\ e_2\ \textbf{else}\ 0
\end{array}$$

typecase *loc* **of** ... $(id_0, id_1, \ldots, id_{n-1})$ **=>** *e* ... $\longrightarrow$ $e \cdot \{v_0/id_0, v_1/id_1, \ldots, v_{n-1}/id_{n-1}\}$

where v_i is the value at the i^{th} index of the array stored at *loc*

typecase v **of** *label id* **=>** *e* ... $\longrightarrow$ $e{\cdot}\{v/id\}$

where *label* is one of **int**, **fun**, or **any** that matches v. If multiple labels match v, the first matching label in the sequence is used. If v is a location, it can be matched to **any** if the case of the form $(id_0, id_1, \ldots, id_{n-1})$ is either absent or occurs after the **any** case in the sequence.

The rules for the memory accesses are as follows:

loc $[v_1]$ $\longrightarrow$ v_2

where *loc* is a location in the global memory. The return value is the value at the v_1^{th} index of the array stored at location *loc*.

array v $\longrightarrow$ *loc*

where *loc* is a new location in the global memory
Side effect: a location *loc* is created in the memory, with its contents initialized to an array that contains v in every element.

$(e_0, e_1, \ldots, e_{n-1})$ $\longrightarrow$ *loc*

where *loc* is a new location in the global memory
Side effect: a location *loc* is created in the memory, 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.

loc $[v_1]$ $:=$ v_2 $\longrightarrow$ v_2

where *loc* is a location in the global memory
Side effect: the v_1^{th} index of the array stored at location *loc* is assigned a value of v_2

Finally, the reductions for concurrent constructs are:

lock *loc* **in** e $\longrightarrow$ **locked** *loc* e

where *loc* is a location in global memory that is currently locked by another thread. Effects: The current thread block waits till it can acquire the lock.

$\mathbf{lock}$ *loc* $\mathbf{in}$ $e \longrightarrow$ $\mathbf{locked}$ *loc* e

where *loc* is a location in global memory and is not currently locked. Effects: location *loc* is locked by the current thread

$\mathbf{locked}$ *loc* $v \longrightarrow v$

where *loc* is a location in global memory that is currently locked by the thread. Effects: the thread releases the lock on location *loc*.

$\mathbf{wait}$ *loc*$_1$ $\mathbf{until}$ *loc*$_2$ $\longrightarrow 0$

where *loc*$_1$ and *loc*$_2$ are locations in global memory. Requires: the location *loc*$_1$ is currently locked by the thread. Effects: the current thread is added to queue of threads blocked waiting on *loc*$_2$. When the thread is signaled, it tries to acquire lock *loc*$_1$ and continue execution.

$\mathbf{signal}$ *loc* $\longrightarrow 0$

where *loc* is a location in global memory. Effects: at least one thread waiting for a signal on location *loc* begins running again.

$\mathbf{do}\ v \longrightarrow e$

where e is the expression returned by the external world
Side effect: send $\mathbf{doAction}(pid,\ v)$ to the external world where pid is the thread identifier. The external world will return the expression e.

$\mathbf{spawn}\ e \longrightarrow n$

Side 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

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

2.6 The external environment

Currently the $\mathbf{do}$ 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:

- $\mathbf{do\ 0}$: reads a number from the input, returns it to the interpreter
- $\mathbf{do\ (1,}\ v\mathbf{)}$: prints the value v to the output and returns v.
- $\mathbf{do\ (2,\ (}c_1\mathbf{,}\ c_2\mathbf{,}\ c_3\mathbf{,}\ \ldots\mathbf{,}\ c_n\mathbf{))}$: prints the characters $c_1, \ldots, c_n$. Returns 1 if well-formatted, 0 otherwise. Here c_i is considered well-formatted if it contains integer expressions.
- $\mathbf{do\ (3,}\ v\mathbf{)}$: if value v is well formed, prints v and returns 1, otherwise prints undefined text and returns 0. Here v is considered well formed if it only contains integer expressions.
- $\mathbf{do\ 4}$: reads a string from the input, returns it to the interpreter

2.7 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, plus a global memory that is shared by all the threads.

We can describe a single thread as a tuple $\langle pid, e\rangle$. The entire interpreter configuration is a triple containing the global memory M, the map W of memory locations to a queue of waiting thread ids and the current queue of threads:

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

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 $e_1 \longrightarrow e_1'$, with effects that change the waiting threads map from W to W' and the global memory M to M'. Then the effect of this step on the configuration as a whole is this:

$$\begin{array}{rl} & \langle M, W,\ \langle pid_1, e_1\rangle, \langle pid_2, e_2\rangle, \ldots, \langle pid_n, e_n\rangle\rangle \\ \longrightarrow & \langle M', W',\ \langle pid_2, e_2\rangle, \ldots, \langle pid_n, e_n\rangle, \langle pid_1, e_1'\rangle\rangle \end{array}$$

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

2.8 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 will be able to communicate with each other if the old thread had allocated locations in the global memory before spawning.

2.9 Errors and termination

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

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

Here, M' is the global memory with all locks belonging to pid_1 released.

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. Of course, any locks that the thread is currently holding are released. Other threads are not directly affected, however. Errors should terminate the thread encountering them, but do not affect other running threads.

2.10 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}\}) \\
(e_1;\ e_2) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & (e_1 \cdot \{\vec{v}/\vec{x}\});\ (e_2 \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{let}\ id = (e_1 \cdot \{\vec{v}/\vec{x}\})\ \mathbf{in}\ (e_2 \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{rec}\ id\ \mathbf{in}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & e{\cdot}\{\vec{v}/\vec{x}\} + \{\mathbf{rec}\ id\ \mathbf{in}\ e/id\} \\
(\mathbf{fn}\ id\ \texttt{=>}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{fn}\ 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}\}) \\
(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{if}\ (e_1 \cdot \{\vec{v}/\vec{x}\})\ \mathbf{then}\ (e_2 \cdot \{\vec{v}/\vec{x}\})\ \mathbf{else}\ (e_3 \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{while}\ e_1\ \mathbf{then}\ e_2) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{while}\ (e_1 \cdot \{\vec{v}/\vec{x}\})\ \mathbf{then}\ (e_2 \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{typecase}\ e_1\ \mathbf{of}\ p\texttt{=>}\ e_2\ \ldots) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{typecase}\ (e_1 \cdot \{\vec{v}/\vec{x}\})\ \mathbf{of}\ p\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}\})) \\
(\mathbf{array}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{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}\}) \\
(\mathbf{lock}\ e_1\ \mathbf{in}\ e_2) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{lock}\ (e_1 \cdot \{\vec{v}/\vec{x}\})\ \mathbf{in}\ (e_2 \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{wait}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{wait}\ (e \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{signal}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{signal}\ (e \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{do}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{do}\ (e \cdot \{\vec{v}/\vec{x}\}) \\
(\mathbf{spawn}\ e) \cdot \{\vec{v}/\vec{x}\} & \longrightarrow & \mathbf{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:

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

3 Using the interpreter

3.1 File structure

The interpreter code is structured as follows:

- **ast/ast.sml:** definitions of basic types (**AST.exp**)
- **eval/memory.sig, memory.sml**: definition of the memory type (**Memory.memory**) and associated operations
- **eval/config.sml:** definition of the configuration type
- **eval/evaluation.sml:** performs a single step of the main interpreter loop. The evaluation searches for the leftmost subexpression to reduce, then calls the reduction function.
- **eval/concurrency.sig, concurrency.sml:** defines all the concurrency operations
- **eval/map.sig, map.sml:** defines the type of memory
- **eval/substitution.sig, substitution.sml** defines the type of the substitution map
- **eval/reductions.sml:** defines the one-step reduction function.
- **eval/gc.sig**, **gc.sml:** garbage collector
- **world/action.sig:** interface for interaction with the external world
- **debug/debug-loop.sml:** interface for debugging
- **cl/*.cl**, a few sample CL programs

3.2 Running CL code

After compiling the code (**CM.make(“sources.cm”)**) 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
- **l** *file*: 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/debug-loop.sml** for more details. If you feel that the debugging tools implemented are inadequate, feel free to modify them.

4 Your task

Part 1: Typecase Evaluation

Finish the implementation of the typecase expession. You will have to make changes to the following files:

- **eval/evaluation.sml**
- **eval/reductions.sml**

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. Gratuitous changes may result in a penalty.

Like 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 SML Basis Library (e.g., arrays and vectors). However, you are *not* allowed to use the additional data structures found in the SML/NJ library, though you may implement your own versions of any data structures if you want to.

You are allowed to modify any code in the **eval** directory. However, you should not modify the **evaluation.sig** file.

To help you figure out where your time is going in the interpreter, you will probably find the SML/NJ profiler (structure **Compiler.Profile** or **Backend.Profile**, depending on which version of SML/NJ you are using) to be helpful. When profiling is turned on with **setProfMode**, SML will record where time is being spent, and can then generate various useful reports. The **reportData** function is one way to get a report. 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.

SML/NJ profiling does not fully work on Windows platforms; you can get the number of times each function was called, but not measurements of time. However, time profiling does work on the Linux version version 110.59 and earlier versions. SML/NJ 110.59 is installed on all the CSUG Linux machines (**empire**, **fuji**, **gala**, **csug01–10**) in **/usr/local/smlnj/bin/sml**. You'll need to add **/usr/local/smlnj/bin** to your path to use it.

Part 3: CL Priority Queue

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 thread-safe priority queue in CL. That is, multiple threads should be able to use your shared memory priority queue concurrently. For example, 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 **lock**.

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 and has them both enqueue and dequeue 1000 elements.

Part 4: 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**.

Part 5: Written problems

There are three written parts to this assignment:

(a) Derive a recurrence relation for $T(h)$ where $T(h)$ is the maximum number of nodes in a tree of height h. The maximum number of children a node is allowed to have is h where h is the height of the subtree with the node as the root. Solve for the closed form of $T(h)$ and prove you answer.

(b) Find an asymptotic upper bound for the following relation, and show it is correct:
$f(n) = k * f(n/2) + n$ for $k > 1$

Files to submit

- **PS5.zip**: A zip file containing all the files to run the interpreter
- **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.