Quandary Language and Runtime Specification

Mike Bond

October 4, 2020

1 Context-free grammar, with notes about semantics

Colors denote productions used only for heap (including the `Q` and `Ref` types), concurrency, and mutation. Later, the list of built-in functions uses the same color coding.

⟨*program*⟩ ::= ⟨*funcDefList*⟩

⟨*funcDefList*⟩ ::= ⟨*funcDef*⟩ ⟨*funcDefList*⟩
| ϵ

⟨*funcDef*⟩ ::= ⟨*varDecl*⟩ `(` ⟨*formalDeclList*⟩ `)` `{` ⟨*stmtList*⟩ `}`

⟨*varDecl*⟩ ::= ⟨*type*⟩ `IDENT` // Variables and functions are immutable by default
| `mutable` form updates

⟨*type*⟩ ::= `int` // 64-bit signed integer
| `Ref`
| `Q` `Q`; or `nil`
ype of `int` and `Ref`

⟨*formalDeclList*⟩ ::= ⟨*neFormalDeclList*⟩
| ϵ

⟨*neFormalDeclList*⟩ ::= ⟨*varDecl*⟩ `,` ⟨*neFormalDeclList*⟩
| ⟨*varDecl*⟩

⟨*stmtList*⟩ ::= ⟨*stmt*⟩ ⟨*stmtList*⟩
| ϵ

⟨*stmt*⟩ ::= ⟨*varDecl*⟩ `=` ⟨*expr*⟩ `;` // Declare and initialize variable
| `IDENT =` ⟨*expr*⟩ `;` // Update to already-declared-and-initialized (mutable) variable
| `if (` ⟨*cond*⟩ `)` ⟨*stmt*⟩
| `if (` ⟨*cond*⟩ `)` ⟨*stmt*⟩ `else` ⟨*stmt*⟩
| `while (` ⟨*cond*⟩ `)` ⟨*stmt*⟩ // Pointless without mutation
| `IDENT (` ⟨*exprList*⟩ `) ;` // `IDENT` must be mutable function
| `free` ⟨*expr*⟩ `;` // Frees memory iff explicit memory management enabled
| `print` ⟨*expr*⟩ `;` // Prints evaluated value followed by a newline
| `return` ⟨*expr*⟩ `;`
| `{` ⟨*stmtList*⟩ `}`

⟨*exprList*⟩ ::= ⟨*neExprList*⟩
| ϵ

⟨*neExprList*⟩ ::= ⟨*expr*⟩ `,` ⟨*neExprList*⟩

```
                | ⟨expr⟩

⟨expr⟩ ::= nil                                          // Special constant value of type Ref
         | INTCONST                                     // 64-bit signed integer of type int
         | IDENT
         | - ⟨expr⟩
         | ( ⟨type⟩ ) ⟨expr⟩                            // Need for explicit downcast from Q to int or Ref
         | IDENT ( ⟨exprList⟩ )
         | ⟨binaryExpr⟩
         | [ ⟨binaryExpr⟩ ]      // Evaluates the left and right sides of the binary expression concurrently
         | ( ⟨expr⟩ )

⟨binaryExpr⟩ ::= ⟨expr⟩ + ⟨expr⟩
                | ⟨expr⟩ - ⟨expr⟩
                | ⟨expr⟩ * ⟨expr⟩
                | ⟨expr⟩ . ⟨expr⟩                       // Evaluates to a Ref referencing a new heap object

⟨cond⟩ ::= ⟨expr⟩ <= ⟨expr⟩
         | ⟨expr⟩ >= ⟨expr⟩
         | ⟨expr⟩ == ⟨expr⟩                             // For comparing int values only
         | ⟨expr⟩ != ⟨expr⟩                             // For comparing int values only
         | ⟨expr⟩ < ⟨expr⟩
         | ⟨expr⟩ > ⟨expr⟩
         | ⟨cond⟩ && ⟨cond⟩
         | ⟨cond⟩ || ⟨c
         | ! ⟨cond⟩
         | ( ⟨cond⟩ )
```


Lexical analysis: An IDENT is a sequence of letters, digits, and un cter is not a digit.

If an INTCONST exceeds the bounds of a 64-bit signed integer, the interpreter's behavior is undefined.

Quandary's syntax is case sensitive.

Quandary allows Java/C/C++-style "block" comments /* like this */

2 Precedence and dangling `else`

Precedence of operators in high-to-low order:

1. Expressions in parentheses (`()`) or brackets (`[]`)
2. `-` used as a unary operator and `(` ⟨*type*⟩ `)` (cast operator)
3. `*`
4. `-` used as a binary operator and `+`
5. `.`
6. `<=`, `>=`, `==`, `!=`, `<`, and `>`
7. `!`
8. `&&` and `||`

All operators are left assocative.

Dangling `else` ambiguity is resolved by matching an `else` with the nearest `if` statement allowed by the grammar.

3 Static type checking

The Quandary interpreter checks the following rules prior to executing the program.

Declarations: A program must not define a function with the same name as another function, including the built-in functions. A program must only call functions defined in the program or built-in functions. A program must define a function named `main` that takes a single argument of type `int`.

A function must not declare a variable with the same name as a variable that has been defined earlier (including as a parameter) in the same or an outer/contiaining lexical scope (demarcated by curly braces, i.e., `{}`, or by being a single conditional statement in an if/else statement or while loop). An expression may only access variables declared within the same or an outer/containing 1exical scope. *Thus for any variable name v at any program point, either v can be accessed or declared, but never both.*

Types and conversions: All ⟨*expr*⟩ evaluation—including function actuals, return values, and `free` statements—must be statically type-checked as much as possible, according to the following type hierarchy:

Upcasts and same-casts are p icit, i.e., ⟨*expr*⟩ ::= `(` ⟨*type*⟩ ⟨*expr*⟩

Function calls must have the same number of actuals as the function defi

Immutability: Variables and functions are *immutable* un n immutable variable must not be the assigned-to variable in an assignment statement, i.e., ⟨*stmt*⟩ ::= `IDENT =` ⟨*expr*⟩ `;`.

An immutable function's body must not contain calls to `mutable` functions (including built-in `mutable` functions).

A call *statement* may only call a `mutable` function.

Miscellaneous: Every function must be statically guaranteed to return a value. The interpreter's static checking may verify this property by simply checking that the function's last statement is a `return` statement (and reporting an error if not).

A function may contain `return` statements that make code statically unreachable. In general, statically unreachable code is not erroneous.

4 Built-in functions

`Q left(Ref r)` – Returns the left field of the object referenced by `r`

`Q right(Ref r)` – Returns the right field of the object referenced by `r`

`int isAtom(Q x)` – Returns `1` if `x` is `nil` or an `int`, and `0` otherwise (it is a `Ref`)

`int isNil(Q x)` – Returns `1` if `x` is `nil`; returns `0` otherwise (it is an `int` or `Ref`)

`mutable int setLeft(Ref r, Q value)` – Sets the left field of the object referenced by `r` to `value`, and returns `1`

`mutable int setRight(Ref r, Q value)` – Sets the right field of the object referenced by `r` to `value`, and returns `1`

`mutable int acq(Ref r)` – Acquires the lock of the object referenced by `r` and returns `1`

`mutable int rel(Ref r)` – Releases the lock of the object referenced by `r` and returns `1`

`int randomInt(int n)` – Returns a random `int` in $[0, n)$

5 Language semantics (including dynamic type checking) and operation of the interpreter

The interpreter executes the defined function called `main` and passes a command-line parameter as `main`'s argument:

```
$ ./quandary
Expected format: quandary [OPTIONS] QUANDARY_PROGRAM_FILE INTEGER_ARGUMENT
Options:
  -gc (MarkSweep|MarkSweepVerbose|RefCount|Explicit|NoGC)
  -heapsize BYTES
BYTES must be a multiple of t
```


The interpreter prints the ret

```
Interpreter returned ((5 . nil) . (-87 . (9 . 3)))
```


Incorrect command-line parameters, including QUANDARY_ d, have undefined behavior (i.e., the interpreter may fail in any way).

Function calls: Function call semantics are pass-by-value.

Order of evaluation: The interpreter evaluates expressions in left-to-right order, i.e., it evaluates the left side of (non-concurrent) binary expressions before the right side, and it evaluates function call actual expressions in left-to-right order.

Binary boolean operators (`&&` and `||`) use short-circuit evaluation.

Dynamic type checking: The interpreter should check executed type downcasts and report a fatal error on a type downcast failure.

`nil` dereference: Calling `left()`, `right()`, `setLeft()`, or `setRight()` with a first argument evaluating to `nil` should cause a fatal run-time error.

Heap mutation: A new heap object's left and right fields are each initialized to an `int` or `Ref` value, and must remain as either an `int` or `Ref` value, respectively, for the duration of the execution. Thus the interpreter should fail with a dynamic type checking error if the `setLeft()` or `setRight()` function attempts to overwrite an `int` slot with a `Ref` value, or a `Ref` slot with an `int` value. This restriction avoids the implementation challenge of performing accesses to a field, which would require updating both the value and associated type metadata atomically.

Memory management: An execution should report an "out of memory" error if and only if the non-freed memory exceeds the specified maximum heap size.

The interpreter potentially supports explicit memory management and mark–sweep and reference counting garbage collection (and optionally others as well, e.g., semi-space). See command-line arguments above.

Explicit memory management only: An execution that accesses a freed object has undefined semantics. An execution that performs double-free on a reference has undefined semantics. An execution that tries to free `nil` has undefined semantics.

Trace-based garbage collection only: An evaluation of an allocation expression ($\langle binaryExpr \rangle$ ::= $\langle expr \rangle$. $\langle expr \rangle$) performs trace-based GC when and only when the non-freed memory exceeds the specified maximum heap size. Trace-based GC frees objects that are transitively unreachable from the roots (functions' local variables and intermediate values). Implementing support for stopping multiple threads at GC-safe points is not required; if trace-based GC is triggered when multiple threads are active, the interpreter has undefined behavior (but ideally it will report an error, to help with debugging).

Concurrency: A concurrently evaluated binary expression `[` $\langle binaryExpr \rangle$ `]` evaluates the left and right child expressions in two new concurrent threads (i.e., thread fork), and waits for both threads to finish (i.e., thread join). Every thread that is not blocked eventually makes progress.

Thread fork and join and lock acquire and release are synchronization operations that induce happens-before edges. Conflicting accesses unordered by happens-before constitute a data race.

An execution of a program with a data race has undefined semantics. An execution in which a thread performs a `rel()` of a lock it does not hold, has undefined semantics.

Error checking: To help with grading, the interpreter *process* should return one of the following error codes as appropriate:

- 0 – success
- 1 – lexical analysis or par
- 2 – static checking error
- 3 – dynamic type checking error

- 4 – `nil` dereference error
- 5 – Quandary heap out-of-memory error

The interpreter script (`quandary`) should print this return code. Specifically, the script should handle executions as follows.

1. For a non-erroneous, terminated program, the script should print the following as its last two lines:

   ```
   Interpreter returned RETURN_VALUE_OF_MAIN
   Quandary process returned 0
   ```

   where `RETURN_VALUE_OF_MAIN` is return value of the Quandary program's `main` function.

   Printing anything or nothing before that is fine.

2. For a non-erroneous, non-terminating execution (e.g., a program execution with an infinite loop),[1] the script should not terminate. Printing anything or nothing is fine.

3. For an execution that should return error code `ERROR_CODE`, the script should print the following as its last line:

   ```
   Quandary process returned ERROR_CODE
   ```

[1] Execution with unbounded call depth has undefined behavior.

Printing anything or nothing before that is fine.

4. For an execution that has undefined behavior, any behavior and output is fine (including uncaught exceptions). An interpreter can safely assume that programs and inputs with undefined behavior will *not* be executed.

If the *interpreter program itself* runs out of stack memory, runs out of heap memory, or allocates too many threads, then behavior is undefined (any behavior is acceptable). For a reasonable Quandary input program, the interpreter should succeed if given enough stack memory, heap memory, and thread count limit.

6 Implementing the interpreter

An interpreter written in Java or C++ should allocate heap objects into raw memory (represented by a primitive array in Java, for example), and assume that raw memory provides only low-level load, store, and compare-and-set operations. When writing the interpreter, use the provided `Heap` class to emulate raw memory.