Quandary Language and Runtime Specification

Mike Bond

August 20, 2019

1 Grammar, with notes about less-obvious semantics

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

$\langle program \rangle$ ::= $\langle funcDefList \rangle$

$\langle funcDefList \rangle$ ::= $\langle funcDef \rangle$ $\langle funcDefList \rangle$
  | ϵ

$\langle funcDef \rangle$ ::= $\langle varDecl \rangle$ `(` $\langle formalDeclList \rangle$ `)` `{` $\langle stmtList \rangle$ `}`

$\langle varDecl \rangle$ ::= $\langle type \rangle$ `IDENT` // Variables and functions are immutable by default
  | `mutable` $\langle type \rangle$ `IDENT` // Mutable vars can be updated; mutable funcs can perform updates

$\langle type \rangle$ ::= `int` // 64-bit signed integer
  | `Cell` // Reference to a heap object with left and right fields of type `Q`
  | `Q` // Super type of `int` and `Cell`

$\langle formalDeclList \rangle$ ::= $\langle neFormalDeclList \rangle$
  | ϵ

$\langle neFormalDeclList \rangle$ ::= $\langle varDecl \rangle$ `,` $\langle neFormalDeclList \rangle$
  | $\langle varDecl \rangle$

$\langle stmtList \rangle$ ::= $\langle stmt \rangle$ $\langle stmtList \rangle$
  | ϵ

$\langle stmt \rangle$ ::= $\langle varDecl \rangle$ `=` $\langle expr \rangle$ `;` // Declare and initialize variable
  | `IDENT =` $\langle expr \rangle$ `;` // Update to already-declared-and-initialized (mutable) variable
  | `if (` $\langle cond \rangle$ `)` $\langle stmt \rangle$
  | `if (` $\langle cond \rangle$ `)` $\langle stmt \rangle$ `else` $\langle stmt \rangle$
  | `while (` $\langle cond \rangle$ `)` $\langle stmt \rangle$ // Pointless without mutation
  | `return` $\langle expr \rangle$ `;`
  | `{` $\langle stmtList \rangle$ `}`

$\langle exprList \rangle$ ::= $\langle neExprList \rangle$
  | ϵ

$\langle neExprList \rangle$ ::= $\langle expr \rangle$ `,` $\langle neExprList \rangle$
  | $\langle expr \rangle$

$\langle expr \rangle$ ::= `nil` // Lone constant value of type `Q`

```
      | INTCONST                                        // 64-bit signed integer of type int
      | IDENT
      | - ⟨expr⟩
      | ( ⟨type⟩ ) ⟨expr⟩                          // Explicit downcast from Q to int or Cell
      | 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 Cell 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⟩ || ⟨cond⟩
        | ! ⟨cond⟩
        | ( ⟨cond⟩ )
```

An `IDENT` is a sequence of letters, digits, and underscores; the first character cannot be 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.

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*⟩ `)` (downcast 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 typing rules

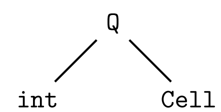
The Quandary interpreter checks the following rules prior to executing the program.

Declarations: A program must not define a function with the same name as another function, including the built-in functions. A function must not declare a variable with the same name as another variable declared in the same function. An expression may only access variables declared within the same or an outer/containing 1exical scope (denoted by curly braces, i.e., {}).

A program must define a function named `main` that takes a single argument of type `int`.

Types and conversions: Function calls must have the same number of actuals as the function definition's number of formals.

All $\langle \mathit{expr} \rangle$ evaluation, including passing function actuals and return values, must be statically type-checked as much as possible, according to the following type hierarchy:



Both silent and explicit upcasts and flat-casts are permitted. Downcasts require an explicit cast ($\langle \mathit{expr} \rangle ::=$ `(` $\langle \mathit{type} \rangle$ `)` $\langle \mathit{expr} \rangle$), which is checked at run time.

Immutability: Variables and functions are *immutable* unless declared as `mutable`. An immutable variable must not be the assigned-to variable in an assignment statement ($\langle \mathit{stmt} \rangle ::=$ `IDENT =` $\langle \mathit{expr} \rangle$ `;`).

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

Miscellaneous: Each function must be statically guaranteed to return a value. The interpreter's static checking may verify this property by simply checking that the interpreter's last statement is a `return` statement (and reporting an error if not). A function may also contain earlier `return` statements, including `return` statements that make code statically unreachable; in general, statically unreachable code is not erroneous.

4 Built-in functions

`Q left(Cell c)` – Returns the left field of the object referenced by `c`

`Q right(Cell c)` – Returns the right field of the object referenced by `c`

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

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

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

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

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

`mutable int rel(Cell c)` – Releases the lock of the object referenced by `c` and returns `1` `int randomInt(int n)` – Returns a random `int` in $[0, n)$

`int free(Cell c)` – Deletes the object referenced by `c` if explicit memory management is supported, and returns `1`; returns `0` otherwise

5 Language semantics 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|RefCount|Explicit)
  -heapsize BYTES
```

The interpreter prints the return value of `main`, e.g.,

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

Function call semantics are pass-by-value.

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

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.

Heap mutation: A heap object's left and right fields are each initialized too an `int` or non-`int` value, and must remain as either an `int` or non-`int` value, respectively, for the duration of the execution. Thus the `setLeft()` and `setRight()` functions should report a run-time error when attempting to overwrite an `int` slot with a non-`int` value, or a non-`int` slot with an `int` value. This restriction avoids the challenge of making atomic accesses to both the values and associated metadata that identifies their types.

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):

```
$ ./quandary
Expected format: quandary [OPTIONS] QUANDARY_PROGRAM_FILE INTEGER_ARGUMENT
Options:
  -gc (MarkSweep|RefCount|Explicit)
  -heapsize BYTES
```

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.

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). If trace-based GC is triggered when multiple threads are active, the interpreter may report a fatal error; implementing support for stopping all threads at GC-safe points is not required.

Concurrency: A concurrently evaluated binary expression ($\langle expr \rangle$ ::= `[` $\langle binaryExpr \rangle$ `]`) 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).

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.

Miscellaneous: If the interpreter program itself runs out of stack memory, runs out of heap memory, or allocates too many threads, it may fail with any error and output. 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.