Magnolisp$_{it}$

Version 6.0.1

Tero Hasu

August 30, 2014

This is a work-in-progress implementation of *Magnolisp*, a small, experimental language and implementation. It is experimental in its implementation technique, which is to replace the phase level 0 (runtime) language of Racket with something non-Racket (here: Magnolisp), and translate it into another language (here: C++) for execution. This variant implementation of Magnolisp—named Magnolisp$_{it}$—uses *identifier tables* for recording annotations.

Magnolisp is an amalgamation of Racket and the likewise experimental programming language Magnolia. Its algebraic language is inspired by Magnolia (or a subset thereof), but adapted for a more natural fit with Racket. Racket provides the module and macro systems. Magnolia is a good fit for C++ translation as it is designed for natural and efficient mapping to most mainstream languages.

1 Magnolisp the Language

```
#lang magnolisp
```
package: magnolisp

The Magnolisp language relies on Racket for its module and macro systems. All of Racket may be used for macro programming. The racket/base language is provided by default for phase level 1 (compile time).

The racket/base definitions (with the exception of the do form) are also available at phase level 0 by default. They may also be used in runtime code, and evaluated as magnolisp. However, only a small subset of Racket can be handled by the Magnolisp compiler, and the compiler will report errors as appropriate for uncompilable language.

When a magnolisp module is evaluated as Racket, any module top-level runtime expressions will also get evaluated; this feature is intended to facilitate testing during development. The Magnolisp compiler, on the other hand, discards top-level expressions, and also any top-level definitions that are not actually part of the program being compiled. One motivation for making most of the racket/base bindings available for runtime code is their potential usefulness in code invoked by top-level expressions.

1.1 Modules and Macros

The Racket provide and require forms may be used as normal, also at phase level 0. However, as far as C++ compilation is concerned, these are only used to connect together Magnolisp definitions internally to the compiled program/library. C++ imports and exports are specified separately using the foreign and export annotations.

For defining macros and doing transformation time computation, the relevant Racket facilities (e.g., define-syntax, define-syntax-rule, begin-for-syntax, etc.) may be used as normal.

1.2 Defining Forms

```
(require magnolisp/surface)
```
package: magnolisp

In Magnolisp, it is possible to declare functions, types (with typedef), and variables; of these, variable definitions are not allowed at the top level. The Magnolisp binding forms are in the magnolisp/surface library. The magnolisp language provides magnolisp/surface at phase level 0.

As Magnolisp has almost no standard library, it is ultimately necessary to define primitive types and functions (flagged as foreign) in order to be able to compile programs that do anything useful.

```
(function (id arg ...) maybe-annos maybe-body)

maybe-body =
           | expr
```

Declares a function. The (optional) body of a function is a single expression, which must produce a single value.

Unlike in Racket, no *tail-call optimization* may be assumed even when a recursive function application appears in *tail position*.

Providing a body is optional in the case where the function is declared as `foreign`, in which case the compiler will ignore any body *`expr`*. When a function without a body is invoked as Racket, the result is `#<void>`. When a `foreign` function with a body is invoked as Racket, the body may be implemented in full Racket, typically to "simulate" the behavior of the C++ implementation. To implement a function body in Racket instead of Magnolisp, enclose the body expression within a `begin-racket` form.

A function with the `export` flag in its annotations indicates that the function is part of the public API of a program that includes the containing module. When a function is used merely as a dependency (i.e., its containing module was not specified as being a part of the program), any `export` flag is ignored.

When a function includes a `type` annotation, the type expression must be of the form *`fn-type-expr`* (see §1.4 "Type Expressions").

For example:

```
(function (identity x)
  x)
(function (five) (#:annos export (type (fn int)))
  5)
(function (inc x) (#:annos foreign (type (fn int int)))
  (add1 x))
(function (seven) (#:annos foreign (type (fn int)))
  (begin-racket 1 2 3 4 5 6 7))
```

Here, `identity` must have a single, concerete type, possible to determine from the context of use. It is not a generic function, and hence it may not be used in multiple different type contexts within a single program.

```
(typedef id maybe-annos)
```

Declares a type. Presently only foreign types may be declared, and *`id`* gives the corresponding Magnolisp name. The `foreign` annotation should always be provided.

For example:

```
(typedef int (#:annos foreign))
(typedef long (#:annos (foreign cxx_long)))
```

```
(var id maybe-annos expr)
```

Declares a local variable with the name *id*, and the (initial) value given by *expr*. A `type` annotation may be included to specify the Magnolisp type of the variable.

For example:

```
> (let ()
    (var x (#:annos (type int)) 5)
    (add1 x))
6
```

```
(let-var id maybe-annos val-expr body)
```

A shorthand for declaring a single, annotated, locally scoped variable. The variable *id* with the initial value given by *val-expr* is only in the scope of the *body* expression. Where no annotations are given, this form is equivalent to `(let ((id val-expr)) body)`. With or without annotations, this form is semantically equivalent to the expression `(do (var id maybe-annos val-expr) (return body))`, provided that *id* does not appear in *val-expr*.

For example:

```
> (let-var x (#:annos (type int)) 5
    (add1 x))
6
```

Where one uses other variants of `let`, it is still possible to specify annotations for the bindings with `anno!`.

1.3 Annotations

```
maybe-annos =
            | (#:annos anno-expr ...)
```

```
         anno-expr = export-anno-expr
                   | foreign-anno-expr
                   | type-anno-expr
                   | ...

  export-anno-expr = export
                   | (export C++-id)

 foreign-anno-expr = foreign
                   | (foreign C++-id)

    type-anno-expr = (type type-expr)
```

where:

C++-id

A valid C++ identifier. When not provided, a default C++ name is automatically derived from the Magnolisp name. For `foreign` declarations, the C++ identifier must naturally match that of an existing C++ definition.

The set of annotations that may be used in Magnolisp is open ended, to allow for additional tools support. Only the most central Magnolisp-compiler-recognized annotations are included in the above grammar.

It is not always necessary to explicitly specify a `type` for a typed Magnolisp definition, as the Magnolisp compiler does whole-program type inference (in Hindley-Milner style). When evaluating as Racket, `type` annotations are not used at all.

For convenience, the `magnolisp` language installs a reader extension that supports annotation related shorthands: `#an(anno-expr ...)` is short for `(#:annos anno-expr ...)`; and `^type-expr` is short for `(type type-expr)`. For example, `#an(^int)` reads as `(#:annos (type int))`.

`(lit-of type-expr literal-expr)`

Annotates a literal, which by themselves are untyped in Magnolisp. While the literal `"foo"` is treated as a `string?` value by Racket, the Magnolisp compiler will expect to determine the literal expression's Magnolisp type based on annotations. The `lit-of` form allows one to "cast" a literal to a specific type for the compiler.

For example:

```
> (lit-of int 5)
5
```

While generally only declarations require annotations, `lit-of` demonstrates a specific case where it is useful to associate annotations with expressions.

```
(anno! id anno-expr ...)
```

Explicitly annotates the identifier *`id`* with the specified annotations. May be used to specify annotations for an identifier that is bound separately, probably by one of the Racket binding forms such as `define`, `let`, etc. May appear in any context in which a `begin` form may appear, and in which the annotated identifier is in scope.

For example:

```
> (let ()
    (define x 5)
    (anno! x (type int))
    x)
5
```

1.4 Type Expressions

```
    type-expr = type-id
              | fn-type-expr

 fn-type-expr = (fn type-expr ... type-expr)
```

Type expressions are parsed according to the above grammar, where *`type-id`* must be an identifier that names a type. The only built-in type is `predicate`, and any others must be declared using `typedef`. The `(fn type-expr ... type-expr)` form contains type expressions for arguments and the return value, in that order. A Magnolisp function always returns a single value.

1.5 Statements and Expressions

Unlike Racket, the Magnolisp language makes a distinction between statements and expressions. Although Magnolisp supports *some* of the Racket language, a given Racket construct must typically appear only in a specific context (either statement or expression context).

In Magnolisp, an `if` form is either a statement or expression, depending on context. That is, depending on context the form is either `(if test-expr then-expr else-expr)` or `(if test-expr then-stat else-stat)`. The `when` and `unless` forms are always statements, and contain statements in their body. The *`test-expr`* conditional expression must always be of type `predicate`, and whether it holds depends on the "truthiness" of its value, as interpreted in C++ or Racket (as applicable).

A `(begin stat ...)` form, in Magnolisp, signifies a sequence of statements, itself constituting a statement. Similarly to Racket, to allow declarations to appear within a statement sequence, `(let () stat ...)` should be used instead.

The `(let ([id expr] ...) body ...+)`, `(let* ([id expr] ...) body ...+)`, and `(letrec ([id expr] ...) body ...+)` forms are statements in Magnolisp, and the *body*s must likewise be statements. The named variant of `let` is not supported. A limited form of `let` is supported in expression context—see `let-var`.

The `(set! id expr)` form is an assignment statement in Magnolisp. The left-hand side expression *id* must be a reference to a bound variable. (The *id* may naturally instead be a transformer binding to an assignment transformer, in which case the form is macro transformed as normal.)

In Magnolisp, `(void)` signifies a statement with no effect. Unlike in Racket, arguments are not allowed. The `(values)` form likewise signifies a statement with no effect, when it appears in a statement position. The two differ only when evaluating as Racket, as the former may only appear in a 1-value context, and the latter in a 0-value context.

The `var`, `function`, and `typedef` declaration forms may appear in a statement position. The same is true of `define` forms that conform to the restricted syntax supported by the Magnolisp compiler.

```
(do stat ...)
```

An *expression block* containing a sequence of statements. As the term implies, an expression block is an expression, despite containing statements. The block must produce a single value by `return`ing it. Control must not reach the end of a block expression—the `return` statement must be invoked somewhere before control “falls out” of the block. The returned value becomes the value of the containing `do` expression.

For example:

```
> (do (void)
      (return 1)
      (return 2))
1
```

```
(return expr)
```

A statement that causes any enclosing `do` block (which must exist) to yield the value of the expression *expr*.

1.6 Predicate Expressions

```
(require magnolisp/prelude)        package: magnolisp
```

A *predicate expression* is simply an expression of type `predicate`, which is the only built-in type in Magnolisp.

The `predicate` type and its associated operations are defined by the `magnolisp/prelude` module, which serves as the runtime library of Magnolisp. The `magnolisp/prelude` names are bound for phase level 0 in the `magnolisp` language.

```
predicate : any/c
```

A built-in type. The "literals" of this type are `true` and `false`. All conditional expressions in Magnolisp are of this type.

```
(TRUE) → #t
(FALSE) → #f
```

The only built-in (primitive) functions in Magnolisp are `TRUE` and `FALSE`, which are both of type `(fn predicate)`. While `TRUE` and `FALSE` are built-in, only Racket implementations are provided; suitable implementations must be provided for C++ as necessary, named `mgl_predicate`, `mgl_TRUE`, and `mgl_FALSE`, respectively. The expression `(TRUE)` is expected to always evaluate to a true value, and `(FALSE)` is expected to always evaluate to a false value; in Racket these evaluate to `#t` and `#f`, respectively.

```
true
false
```

There is also shorthand syntax `true` and `false`; said syntaxes expand to `(TRUE)` and `(FALSE)`, respectively.

1.7 Racket Forms

To use Racket code in a runtime context, you may wrap the code in a form that indicates that the code is only intended for evaluation as Racket. Code so wrapped must be grammatically correct Racket, but not necessarily Magnolisp. The wrapping forms merely switch syntaxes, and have no effect on the namespace used for evaluating the enclosed sub-forms; the surrounding namespace is still in effect. Nesting of the wrapping forms is allowed.

```
(begin-racket Racket-form ...)
```

A Racket expression that is equivalent to writing `(let () Racket-form ...)`. The Magnolisp semantics is to: ignore such forms when at module top-level; treat such forms as no-ops in statement context; and treat them as uncompilable expressions when appearing in an expression position. Uncompilable expressions are acceptable for as long as they are not part of a compiled program, or can be optimized away.

For example:

```
> (function (three) (#:annos foreign (type fn int))
    (begin-racket
      (define x 1)
      (set! x (begin 2 3))
      x))

> (three)
3
```

One use case is to `local-require` a Racket definition into a context where a Magnolisp definition by the same name is being implemented. For example:

```
> (function (equal? x y)
    (#:annos (type (fn int int predicate)) foreign)
    (begin-racket
      (local-require (only-in racket/base equal?))
      (equal? x y)))

> (equal? "foo" "foo")
#t
```

```
(begin-for-racket Racket-form ...)
```

Like `begin-racket`, but equivalent to writing `(begin Racket-form ...)`, and hence not necessarily a Racket expression. Intended particularly for allowing the splicing of Racket definitions into the enclosing context, which is not possible with `begin-racket`.

For example:

```
> (begin-for-racket
    (define six 6)
    (define (one-more x) (let dummy () (+ x 1))))

> (function (eight) (#:annos foreign (type fn int))
    (one-more (one-more six)))

> (eight)
8
```

```
(define-for-racket rest ...)
```

Shorthand for writing `(begin-for-racket (define rest ...))`. Intended for introducing a single binding into the enclosing context, with a definition given in the Racket language.

For example:

```
> (define-for-racket two (begin 1 2))

> (function (four) (#:annos foreign (type fn int))
    (begin-racket (* (begin 1 2) two)))

> (four)
4
```

1.8 Fully Expanded Programs

As far as the Magnolisp compiler is concerned, a Magnolisp program is fully expanded if it conforms to the following grammar.

Any non-terminal marked with the subscript “rkt” is as documented in the “Fully Expanded Programs” section of the Racket Reference. Any non-terminal marked with the subscript “ign” is for language that is ignored by the Magnolisp compiler, but which may be useful when evaluating as Racket. Anything of the form `id`$_{id\text{-}expr}$ is actually a non-terminal like *id-expr*, but for the specific identifier *id*. Form `(`$_{\text{'local-ec} \neq \text{\#f}}$ *sub-form* `...)` means the form `(sub-form ...)` whose syntax object has the property `'local-ec` set to a true value.

```
     module-begin-form = (#%module-begin mgl-modlv-form ...)

         mgl-modlv-form = (#%provide raw-provide-spec_rkt ...)
                        | (#%require raw-require-spec_rkt ...)
                        | submodule-form_rkt,ign
                        | (begin mgl-modlv-form ...)
                        | begin-for-syntax-form_ign
                        | module-level-def
                        | define-syntaxes-form_ign
                        | expr_rkt,ign
                        | in-racket-form_ign

  begin-for-syntax-form = (begin-for-syntax module-level-form_rkt ...)
```

```
define-syntaxes-form = (define-syntaxes (trans-id ...) expr_rkt)

    module-level-def = (define-values (id) mgl-expr)
                     | (define-values (id ...)
                         (#%plain-app values_id-expr mgl-expr ...))

            mgl-expr = id
                     | (#%plain-lambda (id ...) mgl-expr)
                     | (if mgl-expr mgl-expr mgl-expr)
                     | (let-values () mgl-expr)
                     | (letrec-values () mgl-expr)
                     | (letrec-syntaxes+values
                           ([(trans-id ...) expr_rkt,ign] ...)
                           ()
                         mgl-expr)
                     | (let-values ([(id) mgl-expr]) mgl-expr)
                     | (letrec-values ([(id) mgl-expr]) mgl-expr)
                     | (letrec-syntaxes+values
                           ([(trans-id ...) expr_rkt,ign] ...)
                           ([(id) mgl-expr])
                         mgl-expr)
                     | (quote datum)
                     | local-ec-expr
                     | (#%plain-app #%magnolisp_id-expr 'foreign-type)
                     | (#%plain-app id-expr mgl-expr ...)
                     | (#%top . id)
                     | (#%expression mgl-expr)
                     | in-racket-form

                stat = (if mgl-expr stat stat)
                     | (begin stat ...)
                     | (let-values
                           ([(id ...)
                             (#%plain-app values_id-expr mgl-expr ...)]
                            ...)
                         stat ...+)
                     | (letrec-values
                           ([(id ...)
                             (#%plain-app values_id-expr mgl-expr ...)]
                            ...)
                         stat ...+)
                     | (letrec-syntaxes+values
                           ([(trans-id ...) expr_{rkt,ign}] ...)
                           ([(id ...)
                             (#%plain-app values_{id-expr} mgl-expr ...)]
                            ...)
                         stat ...+)
                     | (let-values ([() stat] ...)
                         stat ...+)
                     | (letrec-values ([() stat] ...)
                         stat ...+)
                     | (letrec-syntaxes+values
                           ([(trans-id ...) expr_{rkt,ign}] ...)
                           ([() stat] ...)
                         stat ...+)
                     | (let-values ([(id) mgl-expr] ...)
                         stat ...+)
                     | (letrec-values ([(id) mgl-expr] ...)
                         stat ...+)
                     | (letrec-syntaxes+values
                           ([(trans-id ...) expr_{rkt,ign}] ...)
                           ([(id) mgl-expr] ...)
                         stat ...+)
                     | (set! id mgl-expr)
                     | (#%plain-app values_{id-expr})
                     | (#%plain-app void_{id-expr})
                     | local-ec-jump
                     | (#%expression stat)
                     | in-racket-form

      local-ec-expr = (,local-ec ≠ #f #%plain-app call/ec_{id-expr}
                         (#%plain-lambda (id) stat ...))

      local-ec-jump = (,local-ec ≠ #f #%plain-app id-expr mgl-expr)

            id-expr = id
                     | (#%top . id)
                     | (#%expression id-expr)
```

where:

id

An identifier. Not *the* reserved `#%magnolisp` identifier.

trans-id

An identifier with a *transformer binding*.

datum

A piece of literal data. A `(quote datum)` form is a literal in Magnolisp, and its type must be possible to infer from context.

in-racket-form

Any Racket form that has the syntax property `'in-racket` set to a true value. These are ignored by the Magnolisp compiler where possible, and it is an error if they persist in contexts where they ultimately cannot be ignored. The `begin-racket` and `begin-for-racket` forms are implemented through this mechanism.

local-ec-expr

A restricted form of `call/ec` invocation, which is flagged with the syntax property `'local-ec`. The semantic restriction is that non-local escapes (beyond the enclosing function's body) are not allowed.

local-ec-jump

A restricted form of escape continuation invocation, flagged with the syntax property `'local-ec`. The escape must be local.

(*Warning:* For some of the id$_{id\text{-}expr}$ non-terminals, the current parser actually assumes a direct *id*.)

`#%magnolisp : any/c`

A value binding whose identifier is used to uniquely identify some Magnolisp core syntactic forms. The value of the variable does not matter when compiling an Magnolisp, as it is never used. For purposes of evaluating as Racket, it holds some function that may be applied to any number of arguments, and which produces a single, undefined value.

2 Evaluation

Programs written in Magnolisp can be evaluated in the usual Racket way, provided that the `#lang` signature specifies the language as `magnolisp`. Any module top-level phase level 0 expressions are evaluated, and the results are printed (as for Racket's `#%module-begin`).

3 Compiler API

```
(require magnolisp/compiler-api)        package: magnolisp
```

The Magnolisp implementation includes a compiler targeting C++. The `magnolisp/compiler-api` library provides an API for invoking the compiler.

```
(compile-modules  module-path-v
                  ...
                 [#:relative-to rel-to-path-v])
 → compilation-state?
  module-path-v : module-path?
  rel-to-path-v : (or/c path-string? (-> any) false/c) = #f
(compile-files path-s ...) → compilation-state?
  path-s : path-string?
```

Invoke the compiler front end for analysing a Magnolisp program, whose “entry modules” are specified either as module paths or files. Any specified modules that are not in the `magnolisp` language are effectively ignored, as they do not contain any `export`ed Magnolisp definitions. Both functions return an opaque compilation state object, which may be passed to `generate-files` for code generation.

The optional argument *`rel-to-path-v`* is as for `resolve-module-path`. It is only relevant for relative module paths, and indicates to which path such paths should be considered relative.

Any *`path-s`* is mapped to a `` `(file ,path-s) `` module path, coercing *`path-s`* to a string if necessary.

```
(compilation-state? v) → boolean?
  v : any/c
```

Returns `#t` if *`v`* is a compilation state object (as returned by `compile-modules` or `compile-files`), `#f` otherwise.

```
(generate-files  st
                 backends
                [#:outdir outdir
                 #:basename basename
                 #:out out
                 #:banner banner?])   → void?
  st : compilation-state?
  backends : (listof (cons/c symbol? any/c))
  outdir : path-string? = (current-directory)
  basename : string? = "output"
  out : (or/c #f output-port?) = (current-output-port)
  banner? : boolean? = #t
```

Performs code generation for the program whose intermediate representation (IR) is stored in the compilation state *st*. Code generation is only performed with the specified compiler back ends, and for the specified back end specific file types. For instance, to generate both a C++ header and implementation, you may pass *backends* as `'((cxx (cc hh)))`. The *backends* argument is an association list with one entry per backend. Passing *out* as `#f` causes code generation into (separate) files; otherwise the specified output port is used. When *out* is a true value, the *banner?* argument indicates whether banners (with filenames) should be printed to precede individual output files. When *out* is `#f`, the *outdir* argument specifies the output directory for generated files. The *basename* string is used as the "stem" for output file names.

4 mglc

The compiler can also be invoked via the mglc command-line tool, specifying the program to compile. The tool gets installed by invoking raco setup. (Alternatively you may just run it as ./mglc on Unix platforms.)

To compile a program with mglc, list the source files of the program as arguments; the program will consist of all the functions in the listed files that are annotated with the export flag, as well as any code on which they rely. A number of compiler options affecting compilation behavior may be passed, see mglc --help for a list.

An example invocation would be:

```
  mglc --stdout --banner --cxx my-program.rkt
```

which instructs the compiler to print out C++ code into standard output, with banners, for the program "my-program.rkt".

5 Example Code

For sample Magnolisp programs, see the `"test-*.rkt"` files in the `"tests"` directory of the Magnolisp implementation codebase.

6 License

Except where otherwise noted, the following license applies: