Source-to-Source Compilation via Submodules

Tero Hasu[1] Matthew Flatt[2]

[1]BLDL and University of Bergen

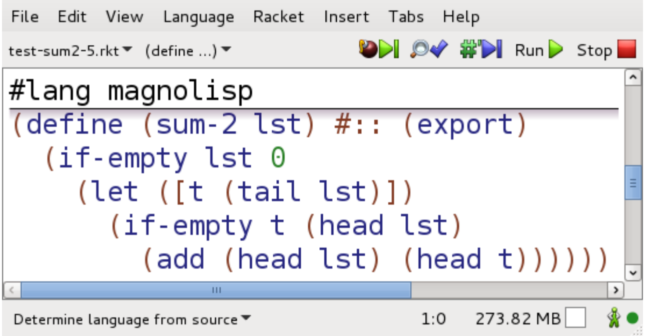
[2]PLT and University of Utah

ELS, 9–10 May 2016

Racket specificity warning

- module-body-transforming macros
  - `#%module-begin`
- complete sub-form expansion
  - `local-expand`
- submodules
  - `module`, `module*`, `module+`

one language environment to rule all targets


File Edit View Language Racket Insert Tabs Help
test-sum2-5.rkt (define ...)
Run
Stop
Determine language from source
1:0
273.82 MB

```
#lang magnolisp
(define (sum-2 lst) #:: (export)
  (if-empty lst 0
    (let ([t (tail lst)])
      (if-empty t (head lst)
        (add (head lst) (head t))))))
```

Racket VM

```
(define-values
  (_sum-2)
  (#%closed sum-220
   (lambda (arg0-785)
     'sum-2 ....
```

C++

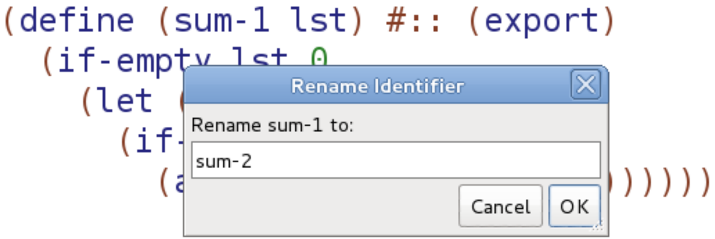
```
MGL_API_FUNC int sum_2(
  List<int> const& lst ) {
  List<int> t;
  return is_empty(lst) ?
    0 : ....
```

motivation for source-to-source compilation

- deploy via a platform-supported language
  - perhaps even *readable* language
    - easier debugging, safer adoption
    - e.g.: Linj, mbeddr, STELLA, PureScript
- use one language to abstract over multiple others
  - e.g.: Haxe, Oxygene, STELLA

motivation for Racket hosting of languages

- stay in Racket's language environment
  - reusing its tools



```
(define (sum-1 lst) #:: (export)
  (if-empty lst 0
    (let (
      (if-
        (a                                      )))))
```

Rename Identifier

Rename sum-1 to: sum-2

- make your language self-extensible
  - macros: lexically scoped, top-level and local, in modules, definition generating, macro generating, in macro implementations, ...

“mouldable” programming

http://mouldable.org/

language support:

- compile-time “concept” implementation composition
- compile-time reasoning about properties and behavior
- compile-time program self-transformations
  - for added convenience and syntactic flexibility

self-extensible languages

- construction with: Lisps, Sugar*, ...?
    - with most “language workbenches”, not so much

```
#lang magnolisp
(define-syntax-rule (if-not c t e)
  (if c e t))
```

Magnolisp

- Rackety syntax
- statically typed, with inference à la Hindley-Milner
- not “functional”—no function values
- runs in Racket, or compiles to C++

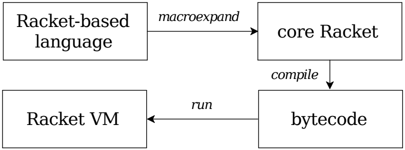
```
#lang magnolisp

(typedef int
  #:: (foreign))

(define (f1 x)
  #:: (export
      ^(-> int int))
  (define (g) x)
  (g))
```

```
MGL_PROTO int f1_g( int const& x );

MGL_API_FUNC int f1( int const& x ) {
  return f1_g(x);
}

MGL_FUNC int f1_g( int const& x ) {
  return x;
}
```

running code within Racket

- #lang line declares the language of a module

```
#lang racket
"Hello World!"
```



Racket-based language

macroexpand

core Racket

compile

bytecode

run

Racket VM

defining languages in Racket

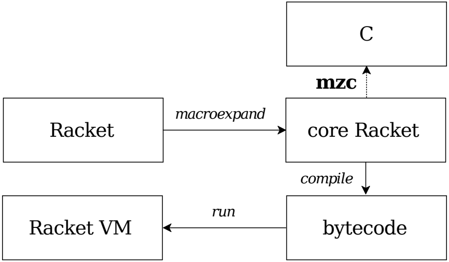
- a `#lang` is implemented as a module
- exports variables, macros, core forms
- specifies a *reader* to turn text into syntax objects

e.g., a `my-lang` just like `racket`:

```
#lang racket
(module reader syntax/module-reader my-lang/main)
(provide (all-from-out racket))
```

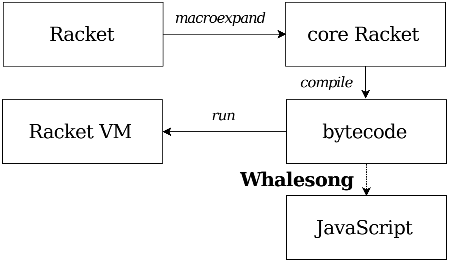
getting known syntax for compilation

- by `read`ing and `expand`ing



C
mzc
Racket
macroexpand
core Racket
compile
Racket VM
run
bytecode

- by parsing bytecode

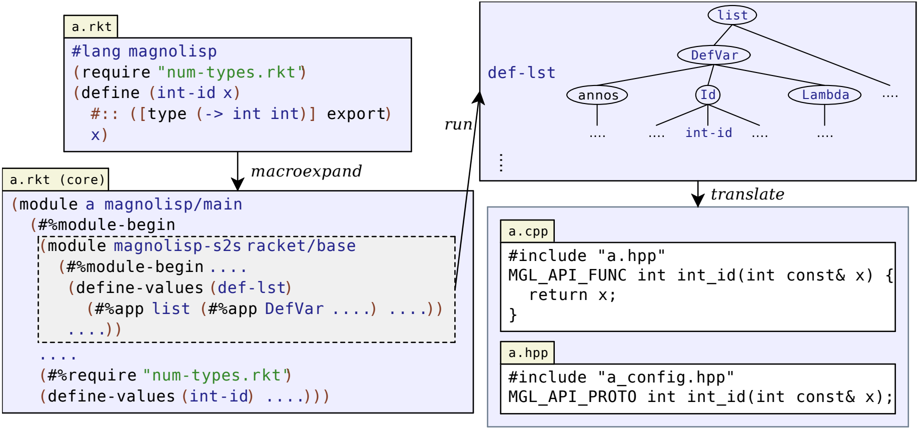


Racket
macroexpand
core Racket
compile
Racket VM
run
bytecode
Whalesong
JavaScript

- by evaluating code as AST constructions
  - e.g., C-Mera

- by treating code as data, and interpreting
  - e.g., SC

or: implement a language that exports syntax



a.rkt

```
#lang magnolisp
(require "num-types.rkt")
(define (int-id x)
  #:: ([type (-> int int)] export)
  x)
```

a.rkt (core)

```
(module a magnolisp/main
  (#%module-begin
   (module magnolisp-s2s racket/base
     (#%module-begin ....
      (define-values (def-lst)
        (#%app list (#%app DefVar ....) ....))
      ....))
   ....
   (#%require "num-types.rkt")
   (define-values (int-id) ....)))
```

a.cpp

```
#include "a.hpp"
MGL_API_FUNC int int_id(int const& x) {
  return x;
}
```

a.hpp

```
#include "a_config.hpp"
MGL_API_PROTO int int_id(int const& x);
```

language getting its own syntax

```
(provide (rename-out [module-begin #%module-begin]))

(define-syntax module-begin
  (λ (stx)
    (do-some-processing-of stx)))
```

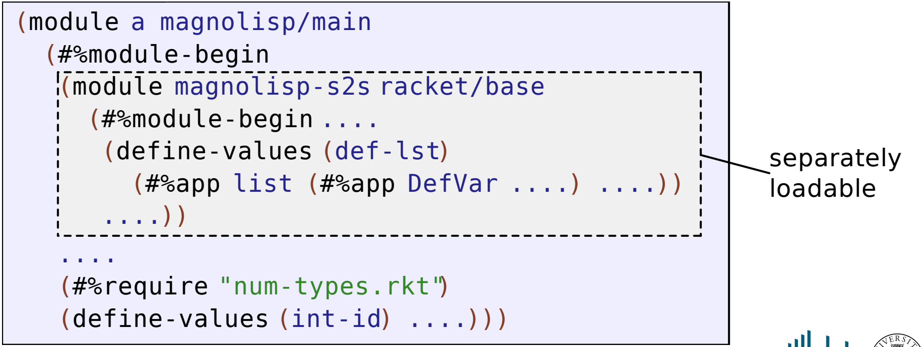
language getting its own core syntax

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
   [(_ . forms)
    (let ([ast (local-expand
               #'(#%module-begin . forms)
               'module-begin '())])
      (do-some-processing-of ast))]))
```

language exporting its own core syntax

have #%module-begin insert a “submodule”

```
(module magnolisp-s2s racket/base
  (require magnolisp/ir-ast)
  (define def-lst (list (DefVar ....) ....)) ....
  (provide def-lst ....))
```



```
(module a magnolisp/main
  (#%module-begin
   (module magnolisp-s2s racket/base
     (#%module-begin ....
      (define-values (def-lst)
        (#%app list (#%app DefVar ....) ....))
      ....))
   ....
   (#%require "num-types.rkt")
   (define-values (int-id) ....)))
```

separately loadable

just a curiosity?

- separate compilation
  - macroexpand and byte-compile only out-of-date modules
- `#lang` itself is in control
  - decides which compilers it supports
  - can, e.g., specify options for compilation

getting the most out of Racket infrastructure

for the hosted language, give:

- Racket-compatible name resolution
- S-expression syntax

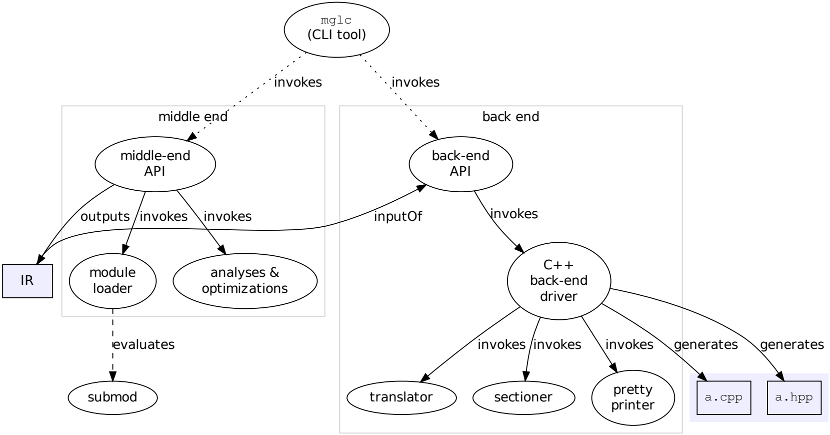
non-Racket core syntax

- Racket expects only *known* core forms and *bound* variable uses

e.g., use a variable binding to identify core-language forms

```
(auto) ↦
(CORE 'auto) ↦
(if #f (#%plain-app #%magnolisp (quote auto)) #f)
```

source-to-source compiler implementation



- Illusyn: term-rewriting strategy combination à la Stratego
  - another alternative for Racket: Nanopass Framework

Magnolisp-based language: Erda$_{C++}$

```
#lang erda/cxx

(require "arith.rkt")

(define (factorial x) #:: (export ^(->Result Int Int))
  #:alert ([bad-arg pre-when (< x 0)])
  (cond
   [(= x 0) 1]
   [else (* x (factorial (- x 1)))]))

5 ;; => (Good 5)
(factorial 5) ;; => (Good 120)
(factorial -5) ;; => (Bad bad−arg)
```

source locations

```
define-values: function return type does not
              match body expression;
  at (source): (f x)
  at (syntax): #<syntax:error-3.rkt:9:2 (#%app f x)>
  in (source): (define (g x)
    #:: (^(-> Int Long) export)
    (f x))
  in (syntax): #<syntax:error-3.rkt:7:0
    (define-values (g) (let-value...>
  declared return type: Long
  actual return type: Int
```

other compile-time mechanisms based on macros

- conditional compilation
- “mapped types”

```
(require (for-syntax "config.rkt"))

(static-cond
 [qt?
  (define-mapped-type String #:mapped-to QString
    [string-index #:mapped-to QString-indexOf]
    ....)]
 [cxx?
  (define-mapped-type String #:mapped-to std::wstring
    [string-index #:mapped-to std::wstring-find]
    ....)])
```

synopsis

approach

Have macros encode foreign core language in terms of Racket's. Implement a #%module-begin to expand and process a module body, and embed an AST-containing submodule for an external compiler.

achieves

Languages (i.e., #lang definitions) getting to decide which compilers they target. Separate macroexpansion and byte compilation.

software and documentation

```
raco pkg install magnolisp
```

contact

tero@ii.uib.no

mflatt@cs.utah.edu