Source-to-Source Compilation in Racket

You Want it in *Which* Language?

Tero Hasu[1] Matthew Flatt[2]

[1]Bergen Language Design Laboratory
University of Bergen

[2]PLT
University of Utah

IFL, 1–3 October 2014

key topics

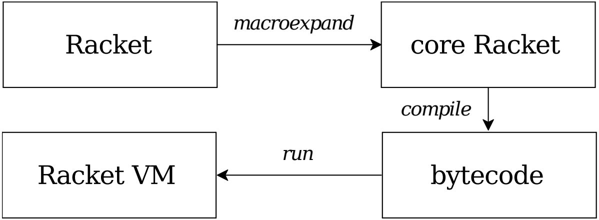
- how to implement source-to-source compilers on top of Racket
- motivations:
  - language infrastructure reuse
  - support for implementing macro-extensible languages

macros for language definition

- Racket macros not only support language *extension*, but also language *definition*
  - host language syntax can be hidden entirely

"normal" execution of Racket languages

- Racket languages are usually executed within the Racket VM



source-to-source compilers

- or *transcompilers*
- programming language implementations outputting source code
- especially nice with exotic platforms
  - have a compiler write what the vendor says you should

don't need no Racket

transcompiler implementation recipe:

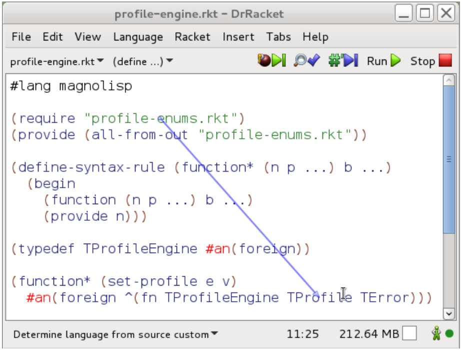
1. pick your favorite programming language
2. pick useful libraries (parsing, pretty printing, etc.)
3. write an implementation

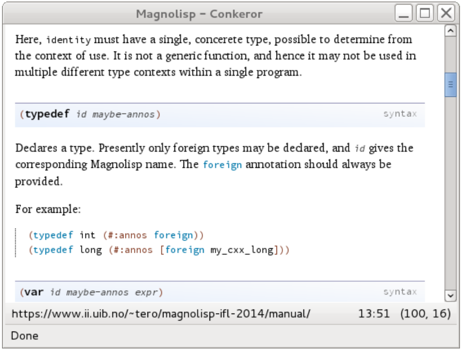
can get back-end side infrastructure reuse

- typically target language libraries
  - e.g., language standard libraries, libuv, OpenGL, SQLite, ...

what about front-end side?

- reuse of language facilities?
  - macro systems, module systems, ...
- reuse of dev tools?
  - IDEs, documentation tools, macro debuggers, ...





language embedding

- can use some host language functionality and tools
  - still syntactically correct language
  - might e.g. get type checking from host

Approaches in Haskell, Scala, etc.:

- shallow embedding
  - language encoded directly as host operations
- deep embedding
  - expressions evaluate to ASTs, which can then be evaluated or translated

language embedding in Racket

- difference: Racket has a compile-time phase built-in
  - gives more options for embedding

An attractive option:

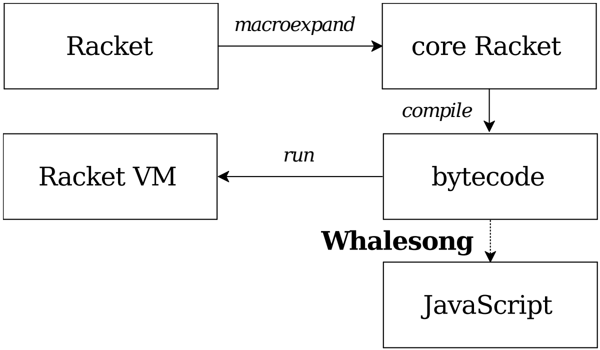
- *macro* expressions evaluate to ASTs, which, still at compile-time:
  - are made to encode Racket VM operations
    - bonus: might write YourLang macros in YourLang
  - are also made available for transcompilation

phase separation

- Racket's *phase separation* guarantees that compile time and run time have distinct bindings and state
- particularly crucial for a transcompiled language
  - run time state: TargetLang (not Racket VM)
  - run time bindings: YourLang (not Racket)

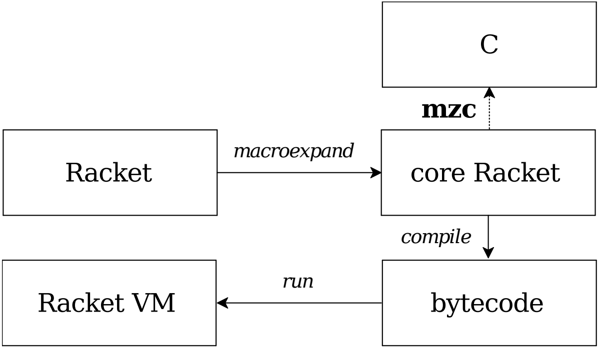
transcompilation via Racket bytecode

- suitable when implementing Racket
- bytecode is optimized for efficiency—does not retain all of the original (core) syntax
- there is an API for parsing bytecode



transcompilation via core Racket

- core syntax for any Racket module can be extracted externally with **read−syntax**, then **expand**
  - `raco expand` has the details



macros in transcompiler implementation

A macro expander *is* a source-to-source "compiler"—macros exist to support source-to-source translation.

- general advantages:
  - macro-based surface syntax definition gives parsing almost "for free"
  - macros are convenient for "sugary" constructs: syntax and semantics specified at once
  - macros are modular and composable

further exploitation of macro-expansion?

- might do back-end-specific work in macro expansion
  - performing target-specific analyses and transformations
  - collating required metadata
  - encoding code and metadata in the desired format
    - made separately loadable, even

Racket submodules

- enable testing time, documentation time, and more
  - adding to Racket's run and compile times

"."	Racket VM run-time code
`main`	code for running the module standalone
`test`	code for testing the module
`srcdoc`	"data-as-code" for inline documentation

can also have:

`to-c++`	code informing a C++ back end
`to-java`	code informing a Java back end

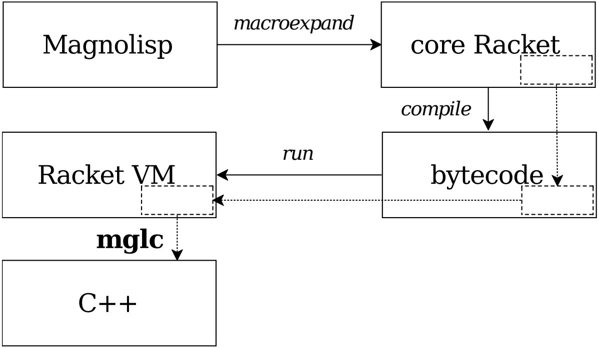
accessing code from within

- a possibility unique(?) to Racket
- a Racket language can access all the code of a module
  - can inspect it unexpanded, or expand it first
  - can munge it in back-end-specific ways

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

compilation based on "transcompile-time" code

- transcompiler **dynamic−require**s a submodule prepared for it during macro expansion
  - e.g. encoding a syntax-checked AST with type annotations



Magnolisp

- a proof-of-concept toy language
- surface syntax defined as macros
- Racket's macro and module systems exposed
  - macro-programming in any Racket VM based language
- execution options:
  1. evaluation in the Racket VM
     - supports "mocking" of primitives, for simulation
  2. by translating runtime code into C++
     - by invoking separate `mglc` tool

Magnolisp syntax sample

```
#lang magnolisp
(typedef Int
  (#:annos foreign))
(function (zero)
  (#:annos foreign [type (fn Int)]))
(function (inc x)
  (#:annos foreign [type (fn Int Int)]))

(function (one)
  (inc (zero)))

(function (two)
  (do (var x (one))
      (return (inc x))))
```

example Magnolisp to C++ translation

```
(function (one)
  (inc (zero)))

(function (two)
  (do (var x (one))
      (return (inc x))))
```

- `mglc` does whole-program optimization, type inference, C++ translation, pretty printing, etc.
- more interesting: the Racket language implementation

```
MGL_FUNC Int one( ) {
  return inc(zero ());
}

MGL_FUNC Int two( ) {
  Int r;
  {
    Int x = one();
    {
      r = inc(x);
      goto b;
    }
  }
  b:
  return r;
}
```

a.rkt

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

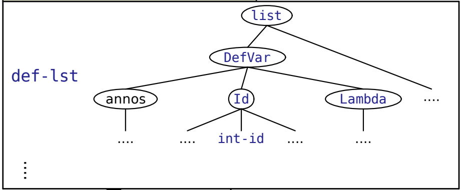
macroexpand

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) ....)))
```

run

a.rkt magnolisp-s2s (instance)



translate

a.cpp

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

a.hpp

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

transcompiled language as a library

- mostly a matter of exporting macros and variables
- syntax should be restricted to what can be transcompiled
- some macros should embed information for transcompilation

E.g., "main.rkt" for `plain-magnolisp` language:

```
#lang racket/base
(module reader syntax/module-reader plain-magnolisp/main)

(require magnolisp/surface)
(provide #%app function typedef foreign export type fn)

(require magnolisp/modbeg)
(provide (rename-out [module-begin #%module-begin]))
```

encoding foreign core language

- a transcompiled language's core language may differ from Racket's
- macros expand to Racket core forms, but:
  - the core forms may have custom syntax properties
  - some variables may have special meaning
  - etc.

E.g., a Magnolisp core form corresponding to a C++ **goto** label, encoded as a **call/ec** application with a specific property:

```
(define-syntax (let/local-ec stx)
  (syntax-case stx ()
    [(_ . rest)
     (syntax-property
      (syntax/loc stx (let/ec . rest))
      'local-ec #t)]))
```

defining surface syntax

- with macros that expand to supported core language

```
(define-syntax-rule
  (do body ...)
  (let/local-ec k
    (syntax-parameterize
     ([return
       (syntax-rules ()
         [(_ v) (apply/local-ec k v)])])
     body ...
     (values))))

(provide do)
```

encoding metadata

- describes a core syntactic construct, but isn't one

```
(function (f x) (#:annos export)
  (g x))
```

encoded as:

```
(define-values (f)
  (let-values ([()
                 (begin
                   (if '#f (#%app #%magnolisp
                                  'anno 'export '#t)
                       '#f)
                   (#%app values))])
    (#%plain-lambda (x) (#%app g x))))
```

where **let−values** has syntax property 'annotate = #t

exporting information for transcompilation

- export in a submodule
  - shift with `begin-for-syntax` as required to prevent running enclosing module upon loading
- encode code as:
  1. syntax-quoted code
     - prevents evaluation, but preserves lexical-binding information
     - as desired, can also preserve source locations or syntax properties
  2. in the IR format used by the compiler
  3. ...

exporting full AST as syntax-quoted code

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(_ form ...)
     (let ([x (local-expand
               #'(#%module-begin form ...)
               'module-begin null)])
       (with-syntax ([(mb . forms) x]
                     [x-lit x])
         #'(mb
            (begin-for-syntax
              (module* to-compile #f
                (provide ast)
                (define ast
                  (quote-syntax/keep-srcloc x-lit))))
            . forms)))]))
```

generality

- a general way to host a transcompiled language in Racket
    - nothing special about Magnolisp
- principal constraint: a binding form in the hosted language must be encoded as a binding form in Racket
    - the process of hygienic macro expansion relies on it
    - in return, Racket resolves names for you, and Racket tools understand binding structure in YourLang

```
(typedef TProfileEngine #an(foreign))

(function* (set-profile e v)
  #an(foreign ^(fn TProfileEngine TProfile TError)))
```

transcompiled-language construction kits

- Rascal
- Spoofax
- Silver
- ...
- Racket

self-extension

A language supports self-extension if the language can be extended by programs of the language itself while reusing the language's implementation unchanged.

Erdweg et al., 2012

language properties allowing pervasive abstraction

Racket supports the definition of languages that have:

1. self-extensibility
   - syntactic extensibility through macros
2. scoping control of extensions
   - module system and local macros
3. safe composition of extensions
   - macro expansion preserves meaning of bindings and references

In other language toolkits, e.g.:

- Sugar* supports (1) and (2)
- Silver supports (3)

conditional compilation (idea)

Use of `#if` & co. is pragmatic in a cross-platform setting.

C++ example:

```
#include "config.hh"

World init_any_ui(World const& w)
{
#if ON_BB10 || ON_HARMATTAN || ON_SAILFISH
  return init_qt_ui(w);
#elif ON_CONSOLE
  return init_ncurses_ui(w);
#else
  return w;
#endif
}
```

conditional compilation (implementation)

```
(define-syntax (static-cond stx)
  (syntax-case stx (else)
    [(_) #'(void)]
    [(_ [else stm]) #'stm]
    [(_ [c stm] . rest)
     (if (syntax-local-eval #'c)
         #'stm
         #'(static-cond . rest))]))
```

where:

- `c` is a Racket conditional expression, evaluated at compile time
- `stm` is a Magnolisp statement, for execution at runtime

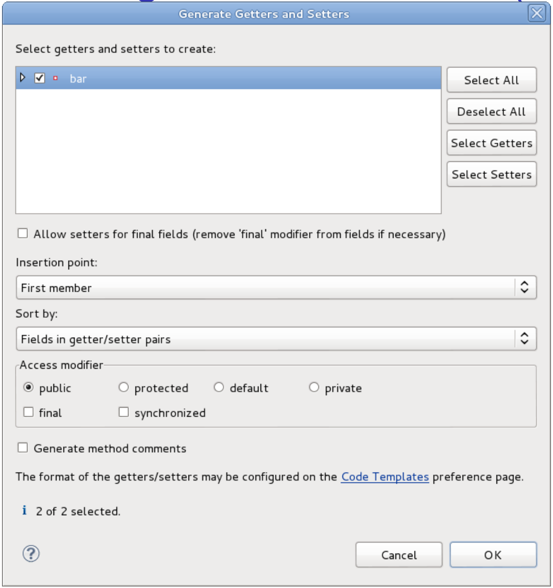
conditional compilation (use)

```
(require (for-syntax "config.rkt"))
(function (init-any-ui w)
  (#:annos export [type (fn World World)])
  (do
    (static-cond
     [(or on-bb10 on-harmattan on-sailfish)
      (return (init-qt-ui w))]
     [on-console
      (return (init-ncurses-ui w))]
     [else
      (return w)])))
```

With `(define on-bb10 #t)`:

```
MGL_API_FUNC World init_any_ui(World const& w) {
  return init_qt_ui(w);
}
```

declaring accessor functions (idea)



declaring accessor functions (implementation)

```
(define-syntax (declare-accessors stx)
  (syntax-case stx ()
    [(_ cls fld t)
     (with-syntax
         ([get (format-id stx "~a-get-~a" #'cls #'fld)]
          [set (format-id stx "~a-set-~a" #'cls #'fld)])
       #'(begin
           (function (get obj)
             (#:annos [type (fn cls t)]
                      foreign))
           (function (set obj v)
             (#:annos [type (fn cls t cls)]
                      foreign))))])))
```

declaring accessor functions (use)

```
(declare-accessors Obj x int)

(function (f obj)
  (#:annos export [type (fn Obj Obj)])
  (Obj-set-x obj (inc (Obj-get-x obj))))
```

```
MGL_API_FUNC Obj f(Obj const& obj)
{
  return Obj_set_x(obj, inc(Obj_get_x(obj)));
}
```

synopsis

A custom source-to-source compiled language can be a Racket language, and it can have Racket's usual scoped and safely composable extensibility from within the language.

proof-of-concept

magnolisp.github.io

contact

tero@ii.uib.no
mflatt@cs.utah.edu