Source-to-Source Compilation via Submodules

Tero Hasu
BLDL and University of Bergen
tero@ii.uib.no

Matthew Flatt
PLT and University of Utah
mflatt@cs.utah.edu

ABSTRACT

Racket's macro system enables language extension and definition primarily for programs that are run on the Racket virtual machine, but macro facilities are also useful for implementing languages and compilers that target different platforms. Even when the core of a new language differs significantly from Racket's core, macros offer a maintainable approach to implementing a larger language by desugaring into the core. Users of the language gain the benefits of Racket's programming environment, its build management, and even its macro support (if macros are exposed to programmers of the new language), while Racket's syntax objects and submodules provide convenient mechanisms for recording and extracting program information for use by an external compiler. We illustrate this technique with Magnolisp, a programming language that runs within Racket for testing purposes, but that compiles to C++ (with no dependency on Racket) for deployment.

CCS Concepts

•Software and its engineering → Extensible languages; Translator writing systems and compiler generators;

Keywords

Language embedding, module systems, separate compilation

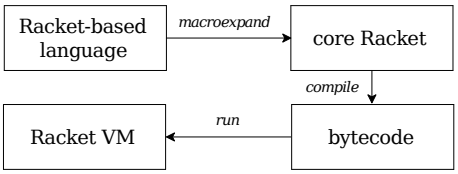
1. INTRODUCTION

A *macro expander* supports the extension of a programming language by translating extensions into a predefined core language. A *source-to-source compiler* (or *transcompiler* for short) is similar, in that it takes source code in one language and produces source code for another language. Since both macro expansion and source-to-source compilation entail translation between languages, and since individual translation steps can often be conveniently specified as macro transformations, a macro-enabled language can provide a convenient platform for implementing a transcompiler.

Racket's macro system, in particular, not only supports language extension—where the existing base language is enriched with new syntactic forms—but also language *definition*—where a completely new language is implemented though macros while hiding or adapting the syntactic forms of the base language. Racket's macro system is thus suitable for implementing a language with a different or constrained execution model relative to the core Racket language.

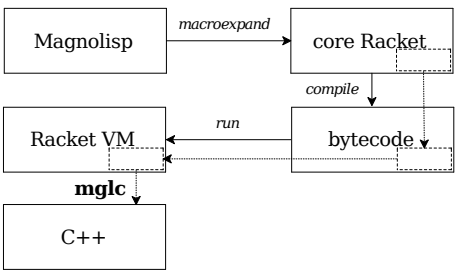
Magnolisp is a Racket-based language that targets embedded devices. Relative to Racket, Magnolisp is constrained in ways that make it more suitable for platforms with limited memory and processors. For deployment, the Magnolisp compiler transcompiles a core language to C++. For development, since cross-compilation and testing on embedded devices can be particularly time consuming, Magnolisp programs also run directly on the Racket virtual machine (VM) using libraries that simulate the target environment.

Racket-based languages normally target only the Racket VM, where macros expand to a core Racket language, core Racket is compiled into bytecode form, and then the bytecode form is run:



To instead transcompile a Racket-based language, Magnolisp could access the representation of a program after it has been macro-expanded to its core (via the `read` and `expand` functions). Fully expanding the program, however, would produce Racket's core language, instead of Magnolisp's core language. External expansion would also miss out on some strengths of the Racket environment, including automatic management of build dependencies.

Magnolisp demonstrates an alternative approach that takes full advantage of Racket mechanisms to assemble a "transcompile time" view of the program. The macros that implement Magnolisp arrange for a representation of the core program to be preserved in the Racket bytecode form of modules. That representation can be extracted as input to the `mglc` compiler to C++:



In this picture, the smaller boxes correspond to a core-form reconstruction that is only run in transcompile-time mode (as depicted by the longer arrow of the "run" step). The boxes are implemented as

submodules (Flatt 2013), and the core form is extracted by running the submodules instead of the main program modules.

By compiling a source program to one that constructs an AST for use by another compiler layer, our approach is similar to lightweight modular staging in Scala (Rompf and Odersky 2010) or strategies that exploit type classes in Haskell (Chakravarty et al. 2011). Magnolisp demonstrates how macros can achieve the same effect, but with the advantages of macros and submodules over type-directed overloading: more flexibility in defining the language syntax, support for static checking that is more precisely tailored to the language, and direct support for managing different instantiations of a program (i.e., direct evaluation versus transcompilation).

2. MAGNOLISP

Magnolisp[1] is statically typed, and all data types and function invocations are resolvable to specific implementations at compile time. Static typing for Magnolisp programs facilitates compilation to efficient C++, as the static types can be mapped directly to their C++ counterparts. To reduce syntactic clutter from annotations and to help retain untyped Racket's "look and feel," Magnolisp supports type inference à la Hindley-Milner.

Magnolisp's surface syntax is similar to Racket's for common constructs, but it also has language-specific constructs, including ones that do not directly map into Racket core language (e.g., `if-cxx` for conditional transcompilation). Magnolisp uses Racket's module system for managing bindings, both for run-time functions and for macros. An exported C++ interface is defined separately through `export` annotations on function definitions; only `exported` functions are declared in the generated C++ header file.

A Magnolisp module starts with `#lang magnolisp`. The module's top-level can `define` functions, types, and so on. A function marked as `foreign` is assumed to be implemented in C++; it may also have a Racket implementation, given as the body expression, to allow it to be run in the Racket VM. Types are defined only in C++, so they are always `foreign`, and `typedef` can be used to give the corresponding Magnolisp declarations. The `type` annotation is used to specify types for functions and variables, and type expressions can refer to declared type names. The `#::` keyword is used to specify a set of annotations for a definition.

In the following example, `add` is a Magnolisp function of type `(-> Int Int Int)`, i.e., a binary function that computes with values of type `Int`. The `(rkt.+ x y)` expression in the function body is a call to a Racket function from the `racket/base` module to approximately simulate C++ `int`eger addition:

```
#lang magnolisp
(require magnolisp/std/list
         (prefix-in rkt. racket/base))

(typedef Int #:: ([foreign int]))
(define (add x y) ; integer primitive (implemented in C++)
  #:: (foreign [type (-> Int Int Int)])
  (rkt.+ x y))
```

No C++ code is generated for the above definitions, as they are both declared as `foreign`. As in Racket, it is possible to define macros; this pattern-based one defines a new conditional, which uses `magnolisp/std/list` module's `empty?` function:

```
(define-syntax-rule (if-empty lst thn els)
  (if (empty? lst) thn els))
```

For an example with a C++ translation, consider `sum-2`, a function that uses the above definitions to compute the sum of the first two elements of its list argument (or fewer for shorter lists):

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

The transcompiler-generated C++ implementation for the `sum-2` function is the following (apart from minor reformatting):

```
MGL_API_FUNC int sum_2( List<int> const& lst ) {
  List<int> t;
  return is_empty(lst) ?
      0 :
      ((t = tail(lst)),
       (is_empty(t) ? head(lst) :
         add(head(lst), head(t))));
}
```

Figure 1 shows an overview of the Magnolisp architecture, including both the `magnolisp`-defined front end and the `mglc`-driven middle and back ends. Figure 2 illustrates the forms of data that flow through the compilation pipeline. Transcompilation triggers running of `"a.rkt"` module's transcompile-time code, through `magnolisp-s2s` submodule's instantiation by invoking `dynamic-require` to fetch values for certain variables (e.g., `def-lst`); the values describe the code of `"a.rkt"`, and are already in the compiler's internal data format. Any referenced dependencies of `"a.rkt"` (e.g., `"num-types.rkt"`, as indicated by `int`'s binding information) are processed in the same manner, and the relevant definitions are incorporated into the compilation result (i.e., `"a.cpp"` and `"a.hpp"`).

The middle and back ends are accessed either via the `mglc` command-line tool or via the underlying API as a Racket module. In either case, the expected input is a set of modules for transcompilation into C++. The compiler loads any transcompile-time code in the modules and their dependencies. Any module with a `magnolisp-s2s` submodule is assumed to be Magnolisp, but other Racket-based languages may also be used for macro programming or simulation. The Magnolisp compiler effectively ignores any code that is not run-time code in a Magnolisp module.

The program transformations performed by the compiler are generally expressed with term-rewriting strategies. These strategies are implemented by a custom combinator library[2] that is inspired by Stratego (Bravenboer et al. 2008). Syntax trees that are prepared for the transcompilation phase instantiate data types that support the primitive strategy combinators of the combinator library.

The compiler middle end implements whole-program optimization (by dropping unused definitions), type inference, and some simplifications (e.g., removal of condition checks where the condition is constant). The back end implements translation from Magnolisp core to C++ syntax (involving, e.g., lambda lifting), copy propagation, C++-compatible identifier renaming, splitting of code into sections (e.g.: public declarations, private declarations, and private implementations), and pretty printing.

3. TRANSLATED-LANGUAGE HOSTING

Magnolisp is an example of a general strategy for building a transcompiled language within Racket. In this section, we describe some details of that process for an arbitrary transcompiled language L. Where the distinction matters, we use L_R to denote a language that is intended to also run in the Racket VM (possibly with mock implementations of some primitives), and L_C to denote a language that only runs through compilation into a different language.

[1] Available from http://bldl.github.io/magnolisp-els16/

[2] http://bldl.github.io/illusyn/

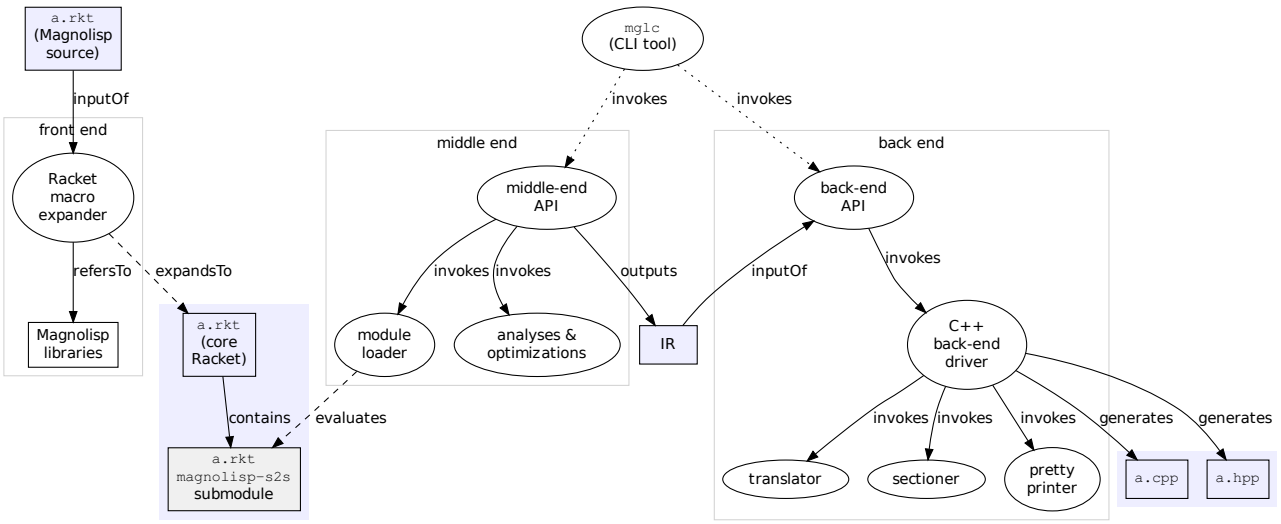


Figure 1: The overall architecture of the Magnolisp implementation, showing some of the components involved in compiling a Magnolisp source file `"a.rkt"` into a C++ implementation file `"a.cpp"` and a C++ header file `"a.hpp"`. The dotted arrows indicate that the use of the `mglc` command-line tool is optional; the middle and back end APIs may also be invoked by other programs. The dashed "evaluates" arrow indicates a conditional connection between the left and right hand sides of the diagram; the `magnolisp-s2s` submodule is *only* loaded when transcompiling. The "expandsTo" connection is likewise conditional, as `"a.rkt"` may have been compiled to bytecode ahead of time, in which case the module is already available in a macro-expanded form; otherwise it is compiled on demand by Racket.

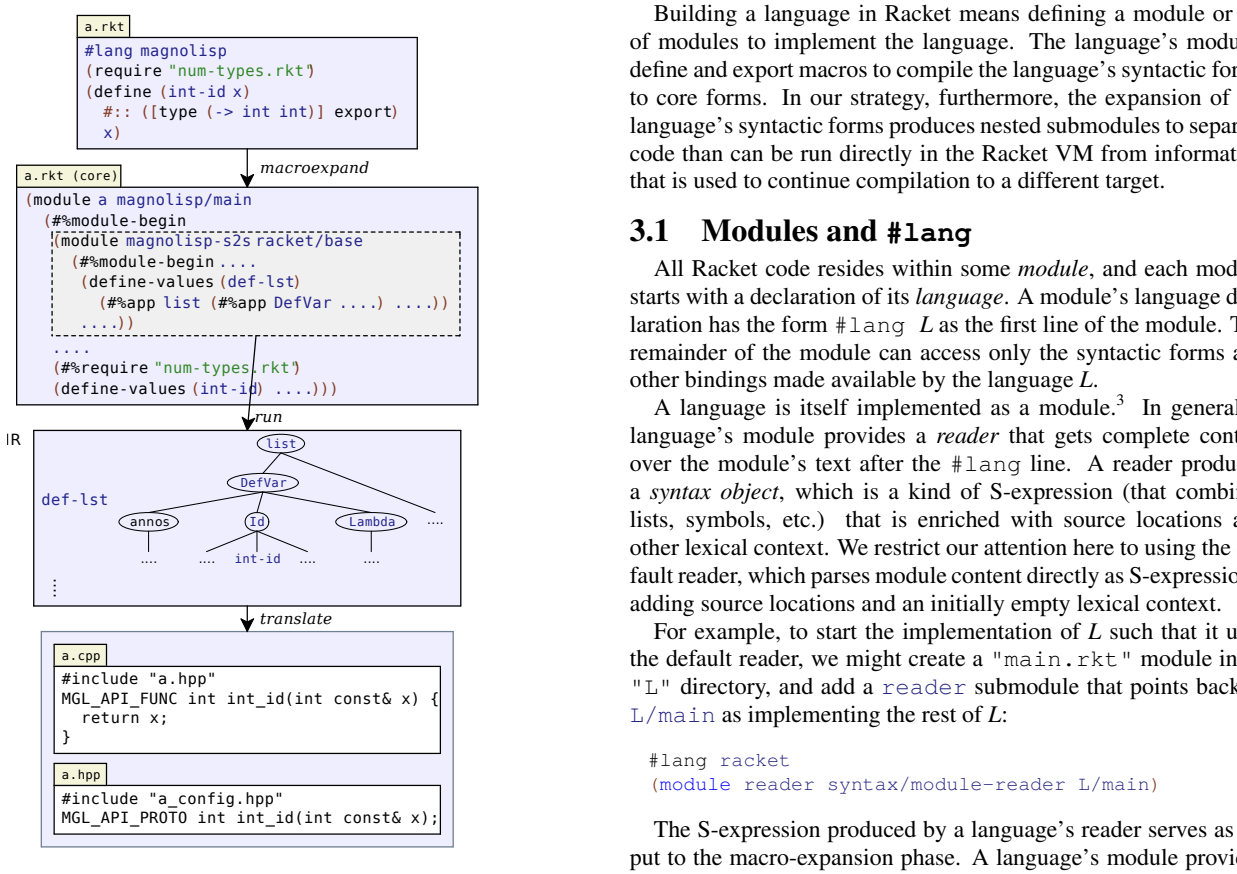


Figure 2: Subset of Figure 1 showing file content: a Magnolisp module passing through the compilation pipeline.

Building a language in Racket means defining a module or set of modules to implement the language. The language's modules define and export macros to compile the language's syntactic forms to core forms. In our strategy, furthermore, the expansion of the language's syntactic forms produces nested submodules to separate code than can be run directly in the Racket VM from information that is used to continue compilation to a different target.

3.1 Modules and `#lang`

All Racket code resides within some *module*, and each module starts with a declaration of its *language*. A module's language declaration has the form `#lang` *L* as the first line of the module. The remainder of the module can access only the syntactic forms and other bindings made available by the language *L*.

A language is itself implemented as a module.[3] In general, a language's module provides a *reader* that gets complete control over the module's text after the `#lang` line. A reader produces a *syntax object*, which is a kind of S-expression (that combines lists, symbols, etc.) that is enriched with source locations and other lexical context. We restrict our attention here to using the default reader, which parses module content directly as S-expressions, adding source locations and an initially empty lexical context.

For example, to start the implementation of *L* such that it uses the default reader, we might create a `"main.rkt"` module in an `"L"` directory, and add a `reader` submodule that points back to `L/main` as implementing the rest of *L*:

```
#lang racket
(module reader syntax/module-reader L/main)
```

The S-expression produced by a language's reader serves as input to the macro-expansion phase. A language's module provides syntactic forms and other bindings for use in the expansion phase

[3] Some language must be predefined, of course. For practical purposes, assume that the `racket` module is predefined.

by exporting macros and variables. A language L can re-export all of the bindings of some other language, in which case L acts as an extension of that language, or it can export an arbitrarily restrictive set of bindings.

A language must at least export a macro named `#%module-begin`, because that form implicitly wraps the body of a module. Most languages simply use `#%module-begin` from `racket`, which treats the module body as a sequence of `require` importing forms, `provide` exporting forms, definitions, expressions, and nested submodules, where a macro use in the module body can expand to any of the expected forms. A language might restrict the body of modules by either providing an alternative `#%module-begin` or by withholding other forms. A language might also provide a `#%module-begin` that explicitly expands all forms within the module body, and then applies constraints or collects information in terms of the core forms of the language.

As an example, the following `"main.rkt"` re-exports all of `racket` except `require` (and the related core language name `#%require`), which means that modules in the language L cannot import other modules. It also supplies an alternate `#%module-begin` macro to pre-process the module body in some way:

```
#lang racket
(module reader syntax/module-reader L/main)
(provide
  (except-out (all-from-out racket)
              require #%require #%module-begin)
  (rename-out [L-module-begin #%module-begin]))
(define-syntax L-module-begin ....)
```

For transcompilation, the `#%module-begin` macro plays a key role in our strategy. A Racket language L that is intended for transcompilation is defined as follows:

- L's module exports bindings that define the language's surface syntax, and expand only to transcompiler-supported run-time forms. We describe this step further in section 3.2
- Macros record any additional metadata required for transcompilation. We describe this step further in section 3.3
- The `#%module-begin` macro expands all the macros in the module body. We describe this step further in section 3.4
- After full macro expansion, `#%module-begin` adds externally loadable information about the expanded module into the module. We describe this step further in section 3.5
- Any run-time support for running programs is provided alongside the macros that define the syntax of the language. We describe this step further in section 3.6

The export bindings of L may include variables, and the presence of transcompilation introduces some nuances into their meaning. When the meaning of a variable in L is defined in L, we say that it is a *non-primitive*. When its meaning is defined in the execution language, we say that it is a *primitive*. When the meaning of its appearances is defined by a compiler (or a macro) of L, we say that it is a *built-in*. As different execution targets may have different compilers, a built-in for one target may be a primitive for another.

3.2 Defining Surface Syntax

A module that implements the surface syntax of a language L exports a binding for each predefined entity of L, whether that entity is a built-in variable, a core-language construct, or a derived form. When the core language is a subset of Racket, derived forms obviously should expand to the subset. Where the core of L is a superset of Racket, additional constructs need an encoding in terms of Racket's core forms where the encoding is recognizable after expansion; possible encoding strategies include:

- **E1**. Use a variable binding to identify a core-language form. Use it in an application position to allow other forms to appear within the application form. Subexpressions within the form can be delayed with `lambda` wrappers, if necessary.
- **E2**. Attach information to a syntax object through its *syntax property* table; macros that manipulate syntax objects must then propagate properties correctly.
- **E3**. Store information about a form in a compile-time table that is external to the module's syntax objects.

A caveat for strategies **E2** and **E3** is that syntax properties and compile-time tables are transient, generally becoming unavailable after a module is fully expanded; any information to be preserved must be reflected as generated code in the module's expansion, as discussed in section 3.5. Another caveat of such "out-of-band" storage is that identifiers in the stored data must not be moved out of band too early; a binding form must be expanded before its references are moved so that each identifier properly refers to its binding.

In the case of L_R, the result of a macro-expansion should be compatible with both the transcompiler and the Racket evaluator. The necessary duality can be achieved if the surface syntax defining macros can adhere to these constraints: (**C1**) exclude Racket core form uses that are not supported by the compiler; (**C2**) add any compilation hints to Racket core forms in a way that does not affect evaluation (e.g., as custom syntax properties); and (**C3**) encode any compilation-specific syntax in terms of core forms that appear only in places where they do not affect Racket execution semantics.

Where constraints **C1**–**C3** cannot be satisfied, a fallback is to have `#%module-begin` rewrite the run- or transcompile-time code (or both) to make it conform to the expected core language. Rewriting may still be constrained by the presence of binding forms.

For cases where a language's forms do not map neatly to Racket binding constructs, Racket's macro API supports explicit *definition context*s (Flatt et al. 2012), which enable the implementation of custom binding forms that cooperate with macro expansion.

For an example of foreign core form encoding strategy **E1**, consider an L_C with a `parallel` construct that evaluates two forms in parallel. This construct might be defined simply as a "dummy" constant, recognized by the transcompiler as a specific built-in by its identifier, translating any appearances of `(parallel e1 e2)` "function applications" appropriately:

```
(define parallel #f)
```

Alternatively, as an example of strategy **E2**, L_C's `(parallel e1 e2)` form might simply expand to `(list e1 e2)`, but with a `'parallel` syntax property on the `list` call to indicate that the argument expressions are intended to run in parallel:

```
(define-syntax (parallel stx)
  (syntax-case stx ()
    [(parallel e1 e2)
     (syntax-property #'(list e1 e2)
                      'parallel #t)]))
```

For L_R, `parallel` might instead be implemented as a simple pattern-based macro that wraps the two expressions in `lambda` and passes them to a `call-in-parallel` run-time function, again in accordance to strategy **E1**. The `call-in-parallel` variable could then be treated as a built-in by the transcompiler and implemented as a primitive for running in the Racket VM:

```
(define-syntax-rule (parallel e1 e2)
  (call-in-parallel (lambda () e1) (lambda () e2)))
```

For an example of adhering to constraint **C3**, we give a simplified definition of Magnolisp's `typedef` form. A declared type `t` is bound as a variable to allow Racket to resolve type references; these bindings also exist for evaluation as Racket, but they are never referenced at run time. The `#%magnolisp` built-in is used to encode the meaning of the variable, but as it has no useful definition in Racket, evaluation of any expressions involving it is prevented. The `CORE` macro is a convenience for wrapping `(#%magnolisp ....)` expressions in an `(if #f .... #f)` form to "short-circuit" the overall expression and make it obvious to the Racket bytecode optimizer that the enclosed expression is never evaluated. The `annotate` form is a macro that stores the annotations `a ...`, which might, for example, include `t`'s C++ name.

```
(define #%magnolisp #f)
(define-syntax-rule (CORE kind arg ...)
  (if #f (#%magnolisp kind arg ...) #f))

(define-syntax-rule (typedef t #:: (a ...))
  (define t
    (annotate (a ...) (CORE 'foreign-type))))
```

3.3 Storing Metadata

A language implementation may involve *metadata* that describes a syntax object, but is not itself a core syntactic construct in the language. Such data may encode information (e.g., optimization hints) that is meaningful to a compiler or other kinds of external tools. Metadata might be collected automatically by the language infrastructure (e.g., source locations in Racket), it might be inferred by macros at expansion time, or it might be specified as explicit *annotations* in source code (e.g., Magnolisp functions' `export`).

Metadata differs from language constructs in that it does not tend to appear (or at least not remain) as a node of its own in a syntax tree. A workable strategy for retaining any necessary metadata is to have *L*'s syntactic forms store it during macro expansion. Encoding strategies **E1**–**E3** apply also for metadata, for which storage in syntax properties is a typical choice. Typed Racket, for example, stores its type annotations in a custom `'type-annotation` syntax property (Tobin-Hochstadt et al. 2011).

Compile-time tables are another likely option for metadata storage. For storing data for a named definition, one might use an *identifier table*, which is a dictionary data structure where each entry is keyed by an identifier. An *identifier*, in turn, is a syntax object for a symbol. Such a table is suitable for both local and top-level bindings, because the syntax object's lexical context can distinguish different bindings that have the same symbolic name.

Recording metadata in compile-time state has the specific advantage of the data getting collated already during macro expansion which enables lookups across macro invocation sites, without any separate program analysis phase. One could, for example, keep track of variables annotated as `#:mut`able, perhaps to enforce legality of assignments already at macro-expansion time, or to declare immutable variables as `const` in C++ output:

```
(define-for-syntax mutables (make-free-id-table))

(define-syntax (my-define stx)
  (syntax-case stx ()
    [(_ x v)
     #'(define x v)]
    [(_ #:mut x v)
     (free-id-table-set! mutables #'x #t)
     #'(define x v)]))
```

It is also possible to encode annotations in the syntax tree proper, which has the advantage of fully subjecting annotations to macro expansion. Magnolisp adopts this approach for its annotation recording, using a special `'annotate`-property-flagged `let-values` form to contain annotations. Each contained annotation expression `a` (e.g., `[type ....]`) has its Racket evaluation prevented by encoding it as a Magnolisp `CORE` form:

```
(define-syntax-rule (type t) (CORE 'anno 'type t))

(define-syntax (annotate stx)
  (syntax-case stx ()
    [(_ (a ...) e)
     (syntax-property
      (syntax/loc stx ; retain stx's source location
        (let-values ([() (begin a (values))] ...)
          e))
      'annotate #t)]))
```

The `annotate`-generated `let-values` forms introduce no bindings, and their right-hand-side expressions yield no values; only the expressions themselves matter. Where the annotated expression `e` is an initializer expression, the Magnolisp compiler decides which of the annotations to actually associate with the initialized variable.

3.4 Expanding Macros

One benefit of reusing the Racket macro system with *L* is to avoid having to implement an *L*-specific macro system. When the Racket macro expander takes care of macro expansion, the remaining transcompilation pipeline only needs to understand *L*'s core syntax (and any related metadata). Racket includes two features that make it possible to expand all the macros in a module body, and afterwards process the resulting syntax, all within the language.

The first of these features is the `#%module-begin` macro, which can transform the entire body of a module. The second is the `local-expand` (Flatt et al. 2012) function, which may be used to fully expand all the `#%module-begin` sub-forms.

The `local-expand` operation also supports *partial* sub-form expansion, as it takes a "stop list" of identifiers that prevent descending into sub-expressions with a listed name. At first glance, one might imagine exploiting this feature to allow foreign core syntax to appear in a syntax tree, and simply prevent Racket from proceeding into such forms. That strategy would mean, however, that foreign binding forms would not be accounted for in Racket's binding resolution. It would also be a problem if foreign syntactic forms could include Racket syntax sub-forms, as such sub-forms would need to be expanded along with enclosing binding forms.

3.5 Exporting Information to External Tools

After the `#%module-begin` macro has fully expanded the content of a module, it can gather information about the expanded content to make it available for transcompilation. The gathered information can be turned into an expression that reconstructs the information, and that expression can be added to the overall module body that is produced by `#%module-begin`.

The information-reconstructing expression should *not* be added to the module as a run-time expression, because extracting the information for transcompilation would then require running the program (in the Racket VM). Instead, the information is better added as compile-time code. The compile-time code is then available from the module while compiling other *L* modules, which might require extra compile-time information about a module that is imported into another *L* module. More generally, the information can be extracted by running only the compile-time portions of the module, instead of running the module normally.

As a further generalization of the compile versus run time split, the information can be placed into a separate *submodule* within the

module. A submodule can have a dynamic extent (i.e., run time) that is unrelated to the dynamic extent of its enclosing module, and its bytecode may even be loaded separately from that of the enclosing module. As long as a compile-time connection is acceptable, a submodule can include syntax-quoted data that refers to bindings in the enclosing module, so that information can be easily correlated with bindings that are exported from the module.

For example, suppose that *L* implements definitions by producing a normal Racket definition for running within the Racket virtual machine, but it also needs a syntax-quoted version of the expanded definition to compile to a different target. The `module+` form can be used to incrementally build up a `to-compile` submodule that houses definitions of the syntax-quoted expressions:

```
(define-syntax (L-define stx)
  (syntax-case stx ()
   [(L-define id rhs)
    (with-syntax ([rhs2 (local-expand #'rhs
                         'expression null)])
     #'(begin
         (define id rhs2)
         (begin-for-syntax
           (module+ to-compile
             (define id #'rhs2)))))]))
```

The `begin-for-syntax` wrapping makes the `to-compile` submodule reside at compilation time relative to the enclosing module, so that loading the submodule will not run the enclosing module. Within `to-compile`, the expanded right-hand side is quoted as syntax using `#'`. Syntax-quoted code is often a good choice of representation for code to be compiled again to a different target language, because lexical-binding information is preserved in a syntax quote. Source locations are also preserved, so that a compiler can report errors or warnings in terms of a form's original location (`mglc` fetches original source text based on location).

Another natural representation choice is to use any custom intermediate representation (IR) of the compiler. Magnolisp, for example, processes Racket syntax trees already during macro expansion, turning them into its IR format, which also incorporates metadata. The IR uses Racket `struct` instances to represent AST nodes, while still retaining some of the original Racket syntax objects as metadata, for purposes of transcompile-time reporting of semantic errors. Magnolisp programs are parsed at least twice, first from text to Racket syntax objects by the reader, and then from syntax objects to the IR by `#%module-begin`; additionally, any macros effectively parse syntax objects to syntax objects. As parsing is completed already in `#%module-begin`, any Magnolisp syntax errors are discovered even when just evaluating programs as Racket.

The `#%module-begin` macro of `magnolisp` exports the IR via a submodule named `magnolisp-s2s`. The submodule contains an expression that reconstructs the IR, albeit in a somewhat lossy way, excluding details that are irrelevant for compilation. The IR is accompanied by a table of identifier binding information indexed by module-locally unique symbols, which the transcompiler uses for cross-module resolution of top-level bindings, to reconstruct the identifier binding relationships that would have been preserved by Racket if exported as syntax-quoted code. As `magnolisp-s2s` submodules do not refer to the bindings of the enclosing module, they are loadable independently from it.

3.6 Run-Time Support

The modules that implement a Racket language can also define run-time support for executing programs. For *L*, such support may be required for the compilation target environment; for L_R, any support would also be required for the Racket VM. Run-time support for *L* is required when *L* exports bindings to run-time variables, or when the macro expansion of *L* can produce code referring to run-time variables (even if such a variable's run-time existence is very limited, as it is for `#%magnolisp`).

Every run-time variable requires a run-time binding, to make it possible for Racket to resolve references to them. When binding built-ins and primitives of L_C, any initial value expression can be given, as the expressions are not evaluated. A literal constant expression is a suitable initializer for built-ins of L_R, which are initialized for Racket VM execution, but generally never referenced.

Each non-primitive is—by definition—implemented in *L*, with a single definition applicable for all targets. Strictly speaking, though, any non-primitive that is *exported* by a Racket module *L cannot* itself be implemented in *L*, but must use a smaller language; the Racket module system does not allow cyclic dependencies.

Defining a primitive of *L* involves specifying a translation for appearances of the variable into any target language. For a Racket VM target, the variable's value must specify its meaning. For other targets, it may be most convenient to specify the target language mapping in *L*, assuming that *L* includes specific language for that purpose. As the mappings are only needed during transcompilation, any metadata specifying them might be placed into a module that is only loaded on demand by the compiler.

The `magnolisp` language, for example, binds three run-time variables, all of which are built-ins. Of these, `#%magnolisp` is only used for its binding, and only during macro expansion. The compiler knows that conditional expressions must always be of type `Bool`, and that `Void` is *the* unit type of the language; this knowledge is useful during type checking and optimization. References to the Magnolisp built-ins may appear in code generated by `magnolisp`'s macros, and hence they must already be bound for the language implementation. Their metadata (specifying C++ translations) is not required by the macros, however, which makes it possible to `declare` that information separately, using Magnolisp's own syntax for storing metadata for an existing binding:

```
#lang magnolisp/base
(require "core.rkt" "surface.rkt")
(declare #:type Bool #:: ([foreign bool]))
(declare #:type Void #:: ([foreign void]))
```

4. EVALUATION

Our Racket-hosted transcompilation approach is generic—in theory capable of accommodating a large class of languages. In practice, we imagine that it is most useful for hosting newly developed languages (such as Magnolisp), where design choices can achieve a high degree of reuse of the Racket infrastructure. In particular, Racket's support for creating new, extensible languages could be a substantial motivation to follow our approach. Racket hosting is particularly appropriate for an evolving language, since macros facilitate quick experimentation with language features.

Another potential use of our strategy is to add transcompilation support for an existing Racket-based language. We have done so for Erda[4], creating $Erda_{C++}$ as its C++-translatable variant. Erda has Racket-like syntax, but its evaluation differs significantly from both Racket and Magnolisp. Erda_{C++} programs nonetheless compile to C++ using an unmodified Magnolisp compiler.

Erda_{C++} illustrates that Magnolisp is not only a language, but also infrastructure for making Racket-based languages translatable into C++. A Magnolisp-based language must be transformable into Magnolisp's core language, which is more limited than that of Racket (lacking first-class functions, escaping closures, etc.), but the language can have its own runtime libraries (whose names

[4]http://bldl.github.io/erda/

must be `magnolisp-s2s`-communicated to `mglc`). The Racket API of Magnolisp includes a `make-module-begin` function that makes it convenient for other languages to implement `mglc`-compatible `#%module-begin` macros—ones that communicate all the expected information.

A potential drawback of transcompilation is the disconnect between the original, unexpanded code and its corresponding generated source code, which can lead to difficulties in debugging. The problem is made worse by macros, and it can be particularly pressing when the output is hard for humans to read. As Racket's macro expansion preserves source locations, a transcompiler could at least emit the original locations via `#line` directives (as in C++) or source maps (as supported by some JavaScript environments).

4.1 Language Design Constraints

In our experience, two design constraints make Racket reuse especially effective: the hosted language's name resolution should be compatible with Racket's, and its syntax should use S-expressions.

Overloading as a language feature, for instance, appears a bad fit for Racket's name resolution. Instead of overloading, names in Racket programs are typically prefixed with a datatype name, as in `string-length` and `vector-length`. Constructs for renaming at module boundaries, such as `prefix-in` and `prefix-out`, help implement and manage name-prefixing conventions.

An S-expression syntax is not strictly necessary, but Racket's macro programming APIs work especially well with its default parsing machinery. The language implementor can then essentially use concrete syntax in patterns and templates for matching and generating code. This machinery is comparable to concrete-syntax support in program transformation toolkits such as Rascal (Klint et al. 2009) and Spoofax (Kats and Visser 2010). Still, other kinds of concrete syntaxes can be adopted for Racket languages, with or without support for expressing macro patterns in terms of concrete syntax, as demonstrated by implementations of Honu (Rafkind and Flatt 2012) and Python (Ramos and Leitão 2014).

5. RELATED WORK

Many language implementations run on Lisp dialects and also target other environments. Some languages, such as Linj (2013) or Clojure plus ClojureScript (2016), primarily provide a Lisp-like language in the target environment. Other languages, such as STELLA (Chalupsky and MacGregor 1999) and Parenscript (2016), primarily match the target environment's semantics but enable execution in a Lisp as well. Magnolisp is closer to the latter group, in that it primarily targets the target environment's semantics.

Most other languages previously implemented on Racket have been meant for execution only on the Racket virtual machine, but a notable exception is Dracula (Eastlund 2012), which compiles macro-expanded programs to ACL2. Its (so far largely undocumented) compilation strategy is to expand syntactic forms to a subset of Racket's core forms, and to specially recognize applications of certain functions (such as `make-generic`) for compilation to ACL2. The part of a Dracula program that runs in Racket is expanded normally, while the part to be translated to ACL2 is recorded in a submodule through a combination of structures and syntax objects, where binding information in syntax objects helps guide the translation.

Whalesong (Yoo and Krishnamurthi 2013) and Pycket (Bauman et al. 2015) are both implementations *of* Racket targeting foreign language environments. Their approaches to acquiring fully macro-expanded Racket core language differ from ours. Whalesong compiles to JavaScript via Racket bytecode, which is optimized for efficient execution (e.g., through inlining), but does not retain all of the original (core) syntax; thus, it is not the most semantics-rich starting point for translation into foreign languages. The Pycket compiler instead performs external expansion to get core Racket; it `read`s, `expand`s, and JSON-serializes Racket syntax, in order to pass it over to the RPython meta-tracing framework.

Ziggurat (Fisher and Shivers 2008)—also built on Racket (then PLT Scheme)—is a meta-language system for implementing extensible languages. Its approach allows both for self-extension and transcompilation of languages, with different tradeoffs compared to ours. Ziggurat features hygienic macros that are Scheme-like, but have access to static semantics, as defined for a language through other provided mechanisms; Racket lacks specific support for interleaving macro expansion with custom analysis. Ziggurat's macros may be locally scoped, but not organized into separately loadable modules; Racket allows for both. There is basic safety of macro composition with respect to Ziggurat's own name resolution, but composability of custom static semantics depends on their implementation. Ziggurat includes constructs for defining new syntax object types, while our approach requires encoding "tricks."

Lightweight Modular Staging (LMS) (Rompf and Odersky 2010) is similar to our technique in goals and overall strategy, but leveraging Scala's type system and overload resolution instead of a macro system. With LMS, a programmer writes expressions that resemble Scala expressions, but the type expectations of surrounding code cause the expressions to be interpreted as AST constructions instead of expressions to evaluate. The constructed ASTs can then be compiled to C++, CUDA, JavaScript, other foreign targets, or to Scala after optimization. AST constructions with LMS benefit from the same type-checking infrastructure as normal expressions, so a language implemented with LMS gains the benefit of static typing in much the same way that a Racket-based language can gain macro extensibility. LMS has been used for languages with application to machine learning (Sujeeth et al. 2011), linear transformations (Ofenbeck et al. 2013), fast linear algebra and other data structure optimizations (Rompf et al. 2012), and more.

The Accelerate framework (Chakravarty et al. 2011; McDonell et al. 2013) is similar to LMS, but in Haskell with type classes and overloading. As with LMS, Accelerate programmers benefit from the use of higher-order features in Haskell to construct a program for a low-level target language with only first-order abstractions.

The Terra programming language (DeVito et al. 2013) takes an approach similar to ours, as it adopts an existing language (Lua) for compile-time manipulation of constructs in the run-time language (Terra). Like Racket, Terra allows compile-time code to refer to run-time names in a way that respects lexical scope. Terra is not designed to support transcompilation, and it compiles to binaries via Terra as a fixed core language. Another difference is Terra's emphasis on supporting code generation at run time, while our emphasis is on separation of compile and run times.

CGen (Selgrad et al. 2014) is a reformulation of C with an S-expression-based syntax, integrated into Common Lisp. An AST for source-to-source compilation is produced by evaluating the core forms of CGen; this differs from our approach, where run-time Racket core forms are not evaluated. Common Lisp's `defmacro` construct is available to CGen programs for defining language extensions; Racket's lexical-scope-respecting macros compose in a more robust manner. Racket's macro expansion also tracks source locations, which would be a useful feature for a CGen-like tool. CGen uses the Common Lisp package system to implement support for locally and explicitly switching between CGen and Lisp binding contexts, so that ambiguous names are shadowed; Racket does not include a similar facility, although approximations thereof should be implementable within Racket.

SC (Hiraishi et al. 2007) is another reformulation of C with an S-expression-based syntax. It supports language extensions defined by transformation rules written in a separate, Common Lisp based domain-specific language (DSL). The rules treat SC programs as data, and thus SC code is not subject to Lisp macro expansion (as in our solution) or Lisp evaluation (as in CGen). Fully transformed programs (in the base SC-0 language) are compiled to C source code. SC programs themselves have access to a C-preprocessor-style extension mechanism via which there is limited access to Common Lisp macro functionality.

6. CONCLUSION

We have described a generic approach for having Racket host the front end of a source-to-source compiler. It involves a proper embedding of the hosted language into Racket, so that Racket's usual language definition facilities are exploited rather than bypassed. Notably, the macro and module systems are still available and, if exposed to the hosted language, provide a way to implement and manage language extensions within the language. Furthermore, tools such as the DrRacket IDE work with the hosted language, recognize the binding structure of programs written in the language, and can usually trace the origins of macro-transformed code.

Among the various ways to arrange for a source-to-source compiler to gain access to information about a program, our approach is most appropriate when the language's macros target a specific foreign core language and runtime library and when it is useful to avoid "extra-linguistic mechanisms" (Felleisen et al. 2015) by having the language itself communicate its execution requirements to the outside world. Such communications may be prepared as submodules, which can also contain an AST in the appropriate core language and representation, allowing one source language to support multiple different targets. Racket's separate compilation and build management help limit preparation work to modules whose source files or dependencies have changed.

Racket's macro system is expressive enough that the syntax and semantics of a variety of language constructs can be specified in a robust way. Given that typical macros compose safely, and given that hygiene reduces the likelihood of name clashes and allows macros to be defined privately, pervasive use of syntactic abstraction becomes a realistic alternative to manual or tools-assisted writing of repetitive code. Such abstraction can benefit both the codebase implementing a Racket-based language, as well as programs written in a macro-enabled Racket-based language.

Acknowledgements Carl Eastlund provided information about the implementation of Dracula. Magne Haveraaen, Anya Helene Bagge, and anonymous referees provided useful comments on drafts of this paper. This research has in part been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

Bibliography

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A Tracing JIT For a Functional Language. In *Proc. ACM Intl. Conf. Functional Programming*, 2015.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming* 72(1-2), pp. 52–70, 2008.

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proc. Wksp. Declarative Aspects of Multicore Programming*, 2011.

Hans Chalupsky and Robert M. MacGregor. STELLA - a Lisp-like language for symbolic programming with delivery in Common Lisp, C++ and Java. In *Proc. Lisp User Group Meeting*, 1999.

ClojureScript. 2016. https://github.com/clojure/clojurescript

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A Multi-Stage Language for High-Performance Computing. *ACM SIGPLAN Notices* 48(6), pp. 105–116, 2013.

Carl Eastlund. Modular Proof Development in ACL2. PhD dissertation, Northeastern University, 2012.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *Proc. Summit Advances in Programming Languages*, 2015.

David Fisher and Olin Shivers. Building Language Towers with Ziggurat. *J. Functional Programming* 18(5-6), pp. 707–780, 2008.

Matthew Flatt. Submodules in Racket: You Want it *When*, Again? In *Proc. Generative Programming and Component Engineering*, 2013.

Matthew Flatt, Ryan Culpepper, Robert Bruce Findler, and David Darais. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *J. Functional Programming* 22(2), pp. 181–216, 2012.

Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. Experience with SC: Transformation-based Implementation of Various Language Extensions to C. In *Proc. Intl. Lisp Conference*, pp. 103–113, 2007.

Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 444–463, 2010.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. IEEE Intl. Working Conf. Source Code Analysis and Manipulation*, pp. 168–177, 2009.

Linj. 2013. https://github.com/xach/linj

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *Proc. ACM Intl. Conf. Functional Programming*, 2013.

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proc. Generative Programming and Component Engineering*, 2013.

Parenscript. 2016. https://common-lisp.net/project/parenscript/

Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. Generative Programming and Component Engineering*, pp. 122–131, 2012.

Pedro Ramos and António Menezes Leitão. An Implementation of Python for Racket. In *Proc. European Lisp Symposium*, 2014.

Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. Generative Programming and Component Engineering*, pp. 127–136, 2010.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers Based on Staging. In *Proc. ACM Sym. Principles of Programming Languages*, 2012.

Kai Selgrad, Alexander Lier, Markus Wittmann, Daniel Lohmann, and Marc Stamminger. Defmacro for C: Lightweight, Ad Hoc Code Generation. In *Proc. European Lisp Symposium*, 2014.

Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proc. Intl. Conf. Machine Learning*, 2011.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. *SIGPLAN Not.* 47(6), pp. 132–141, 2011.

Danny Yoo and Shriram Krishnamurthi. Whalesong: Running Racket in the Browser. In *Proc. Dynamic Languages Symposium*, 2013.