Source-to-Source Compilation in Racket

You Want it in *Which* Language?

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.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software; D.3.4 [*Programming Languages*]: Processors

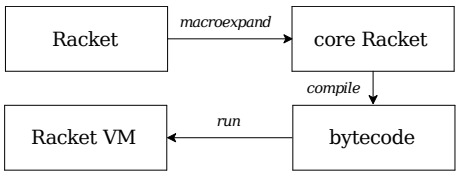
General Terms Design, Languages

Keywords Compiler frameworks, language embedding, macro systems, module systems, syntactic extensibility

1. Introduction

The Racket programming language (Flatt and PLT 2010) builds on the Lisp tradition of language extension through compile-time transformation functions, a.k.a. *macros*. Racket macros not only support 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.

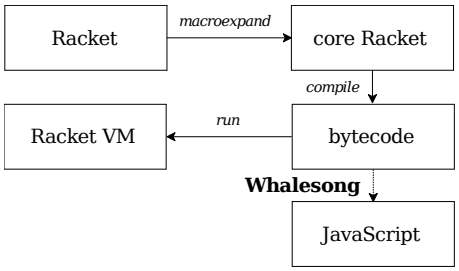
Racket-based languages normally target the Racket virtual machine (VM), where macros expand to a core Racket language, core Racket is compiled into bytecode form, and then the bytecode form is run:



The macro-expansion step of this chain is an example of a *source-to-source compiler* (or *transcompiler* for short), i.e., a translator that takes the source code in one language and outputs source code of another language. Transcompilers have potential benefits compared with compilation to machine code, such as more economical cross-platform application development by targeting widely supported languages, which enables the building of executables with various platform-vendor-provided toolchains.

A Racket-based language can also benefit by avoiding a runtime dependency on the Racket VM. Breaking the dependency can sometimes ease deployment, as the Racket VM is not well supported in every environment. Furthermore, for a mobile "app" to be distributed in an "app store," for example, it is desirable to keep startup times short and in-transit and in-memory footprints low; even in a stripped-down form, Racket can add significantly to the size of an otherwise small installation package. Factors relating to app-store terms and conditions and submission review process may also mean that avoiding linking in additional runtimes may be sensible or even necessary.

One example of an existing source-to-source compiler that avoids the Racket VM is Whalesong (Yoo and Krishnamurthi 2013), which compiles Racket to JavaScript via Racket bytecode:

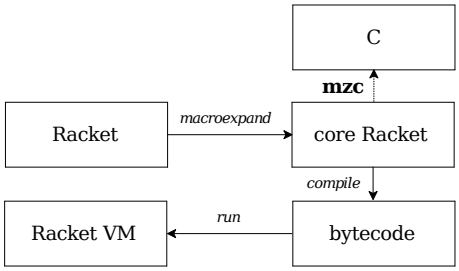


In this approach, a number of optimizations (such as inlining) are performed for bytecode by the normal Racket compiler, making it a sensible starting point for transcompilers that aim to implement variants of Racket efficiently.

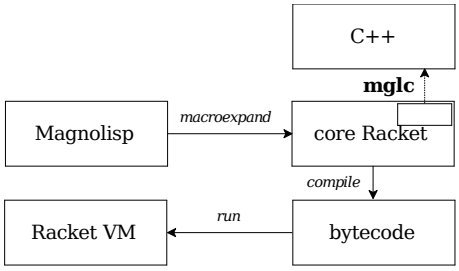
Compiling via Racket bytecode may be less appropriate when the language being compiled is not Racket or when optimizing for properties other than efficiency. Racket's bytecode does not retain all of the original (core) syntax, making it less suitable for implementing semantics-retaining compilation that happens primarily at the level of abstract syntax.

Thus, depending on the context, it may make more sense to compile from macro-expanded core language instead of bytecode.[1] Scheme-to-C compilers (e.g., CHICKEN and Gambit-C) typically work that way, as does the old `mzc` compiler for Racket:

[1] In Racket, one can acquire core language for a Racket source file by `read-syntax`ing the file contents and then invoking `expand` on the read (top-level) forms.



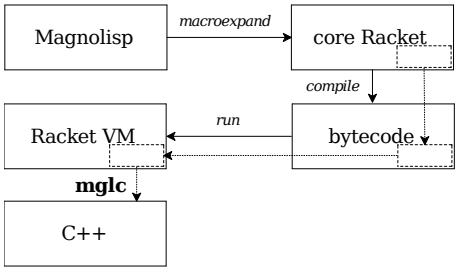
To try out a different Racket-exploiting transcompilation pipeline, we implemented a to-C++ compiler for a small programming language named *Magnolisp*. Conceptually, the Magnolisp implementation is like a Scheme-to-C compiler, but instead of handling all of Racket, it handles only a particular subset of Racket that corresponds to the expansion of the Magnolisp macros (although the "subset" includes additional macro-introduced annotations to guide compilation to C++). A traditional compilation pipeline for `mglc` (the Magnolisp compiler) would be the following, with the smaller box representing the part of the program to be transcompiled (additionally, there can be macro definitions, for example, which are not relevant when transcompiling):



However, directly manipulating core Racket S-expression syntax is not especially convenient from outside of the Racket pipeline. Racket's strength is in support for the macro-expansion phase, especially its support for multiple modules and separate compilation at the module level. It can be useful to be able to do back-end-specific work in macro expansion. In the Magnolisp case, such work includes recording type annotations and catching syntax errors early, for the benefit of `mglc`.

Magnolisp demonstrates a source-to-source compilation approach that takes advantage of the macro-expansion phase to prepare for "transcompile time."[2] More precisely, Magnolisp arranges for macro expansion to embed into the Racket program the information that the Magnolisp compiler needs. The compiler extracts the information by running the program in a mode in which transcompile-time code is evaluated. This results in the following, distinctly non-traditional compilation pipeline; here, the smaller boxes still correspond to the part of the program that is transcompiled, but they now denote code that encodes the relevant information about the program, and only gets run in the transcompile-time mode (as depicted by the longer arrow of the "run" step):

[2] Our use of the word "time" here refers to the idea behind Racket's submodules (Flatt 2013), which is to make it possible for programmers to define new execution phases beyond the macro-expansion and run-time phases that are built into the language model. In our case we want to introduce a transcompile-time phase, and designate some of the code generated during macro expansion as belonging to that phase. In practice, this is done by putting said code into a separately loadable module within a module, i.e. a submodule of a known name within the module whose transcompile-time code it is.



In essence, this strategy works because Racket is already able to preserve syntactic information in bytecode, so that Racket can implement separately compiled macros. Recent generalizations to the Racket syntax system—notably, the addition of submodules (Flatt 2013)—let us conveniently exploit that support for Magnolisp compilation.

The information that Magnolisp makes available (via a submodule) for compilation consists of abstract syntax trees (which incorporate any compilation-guiding annotations), along with some auxiliary data structures. As the particular abstract syntax is only for compilation, it need not conform to the Racket core language semantics; indeed, Magnolisp deviates from Racket semantics where deemed useful for efficient and straightforward translation into C++.

Even for a language that is primarily designed to support transcompilation, it can be useful to also have an evaluator. We envision direct evaluation being useful for simulating the effects of compiled programs, probably with "mock" implementations of primitives requiring functionality that is not available locally. The idea is to gain some confidence that programs (or parts thereof) work as intended before actually compiling them. Cross-compilation and testing on embedded devices can be particularly time consuming; compilation times generally pale in comparison to the time used to transfer, install, and launch a program.

The usual way of getting a Racket-hosted language to evaluate is to macro transform its constructs into the Racket core language, for execution in the Racket VM. Having macro-expansion generate separately loadable transcompile-time code does mean that it could not also generate evaluatable Racket run-time definitions. Magnolisp demonstrates this by supporting both transcompilation and Racket-VM-based evaluation.

1.1 Motivation for Racket-Hosted Transcompiled Languages

Hosting a language via macros offers the potential for extensibility in the hosted language. This means leveraging the host language both to provide a language extension mechanism and a language for programming any extensions. While a basic language extension mechanism (such as the C preprocessor or a traditional Lisp macro system) may be implementable with reasonable effort, safer and more expressive mechanisms require substantial effort to implement from scratch. Furthermore, supporting programmable (rather than merely substitution based) language extensions calls for a compile-time language evaluator, which may not be readily available for a transcompiled language.

Hosting in Racket offers safety and composability of language extensions through lexical scope and phase separation respecting macro expansion. Racket macros' hygiene and referential transparency help protect the programmer from inadvertent "capturing" of identifiers, making it more likely that constructs defined modularly (or even independently) compose successfully. *Phase separation* (Flatt 2002) means that compile time and run time have distinct bindings and state. The separation in the time dimension is particularly crucial for a transcompiler, as it must be possible to parse code without executing it. The separation of bindings, in turn, helps achieve language separation, in that one can have Racket bindings

in scope for compile-time code, and hosted-language bindings for run-time code.

Racket's handling of modules can also be leveraged to support modules in the hosted language, with Racket's `raco make` tool for rebuilding bytecode then automatically serving as a build tool for multi-module programs in the language. The main constraint is that Racket does not allow cycles among module dependencies.

Particularly for new languages it can be beneficial to reuse existing language infrastructure. With a Racket embedding one is in the position to reuse Racket infrastructure on the front-end side, and the target language's infrastructure (typically libraries) on the back-end side. Reusable front-end side language tools might include IDEs (Findler et al. 2002), documentation tools (Flatt et al. 2009), macro debuggers (Culpepper and Felleisen 2007), etc. Although some tools might not be fully usable with programs that cannot be executed as Racket, the run vs. compile time phase separation means that a tool whose functionality does not entail running a program should function fully.

Racket's language extension and definition machinery may be useful not only for users, but also for language implementors. Its macros have actually become a compiler front end API that is sufficient for implementing many general-purpose abstraction mechanisms in a way that is indistinguishable from built-in features (Culpepper and Felleisen 2006). In particular, a basic "sugary" construct is convenient to implement as a macro, as both surface syntax and semantics can be specified in one place.

1.2 Contributions

The main contributions of this paper are:

- we describe a generic approach to replacing the runtime language of Racket in such a manner that information about code in the language can be processed at macro-expansion time, and selectively made available for separate loading for purposes of source-to-source compilation to another high-level language;
- we show that the core language and hence the execution semantics of such a source-to-source compiled language can differ from Racket's;
- we suggest that it may be useful to also make a transcompiled language executable as Racket, and show that this is possible at least for our proof-of-concept language implementation; and
- we show that this approach to language implementation allows Racket's expressive macro and module systems to be reused to make the implemented language user extensible, and to make the scope of language extensions user controllable.

Some of the presented language embedding techniques have been previously used in the implementation of Dracula, to allow for compilation of Racket-hosted programs to the ACL2 programming language; they have remained largely undocumented, however.

The significance of the reported approach beyond the Racket ecosystem is that it supports transcompiler implementation for languages that have all three of the following properties:

- the language is extensible from within itself;
- the scope of each language extension can be controlled separately, also from within the language; and
- there are some guarantees of independently defined extensions composing safely.[3]

[3] In the case of Racket, macros that do not explicitly capture free variables are safe to compose in the limited sense that they preserve the meaning of variable bindings and references during macro expansion (Eastlund and Felleisen 2010).

2. Magnolisp

Magnolisp is a proof-of-concept implementation of a Racket-hosted transcompiled language, and the running example that we use to discuss the associated implementation techniques. As a language, Magnolisp is not exceptional in being suitable for hosting; the techniques described in this paper constitute a general method for hosting a transcompiled language in Racket.

Code and documentation for Magnolisp is available at:

`https://www.ii.uib.no/~tero/magnolisp-ifl-2014/`

2.1 The Magnolisp Language

To help understand the Magnolisp-based examples given later, we give some idea of the syntax and constructs of the language. We assume some familiarity with Racket macro syntax.

Magnolisp is significantly different from Racket in that its overriding design goal is to be amenable to static reasoning; Racket compatibility, for better reuse of Racket's facilities, is secondary. Magnolisp disallows many forms of runtime-resolved dispatch of control that would make reasoning about code harder. Unlike in Racket, all data types and function invocations appearing in programs are resolvable to specific implementations at compile time.

Requiring fully, statically typed Magnolisp programs facilitates compilation to C++, as the static types can be mapped directly to their C++ counterparts. To reduce syntactic clutter due to annotations, and hence to help retain Racket's untyped "look and feel," Magnolisp features whole-program type inference à la Hindley-Milner.

Magnolisp reuses Racket's module system for managing names internally within programs (or libraries), both for run-time names and macros. The exported C++ interface is defined separately through `export` annotations appearing in function definitions; only `export`ed functions are declared in the generated C++ header file.

The presented language hosting approach involves the definition of a Racket language for the hosted language. The Racket language for Magnolisp is named `magnolisp`. At the top-level of a module written in `magnolisp`, one can declare `function`s, for example. A function may be marked `foreign`, in which case it is assumed to be implemented in C++; such a function may also have a Racket implementation, given as the body expression, to also allow for running in the Racket VM. Types can only be defined in C++, and hence 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 the type expressions appearing within may refer to declared type names. The `#:annos` keyword is used to specify the set of annotations for a definition.

In this example, `get-last-known-location` is a Magnolisp function of type `(fn Loc)`, i.e., a function that returns a value of type `Loc`. The `(rkt.get-last-known-location)` expression in the function body might be a call to a Racket function from module `"positioning.rkt"`, simulating position information retrieval:

```
#lang magnolisp
(require (prefix-in rkt. "positioning.rkt"))

(typedef Loc (#:annos foreign))

(function (get-last-known-location)
  (#:annos foreign [type (fn Loc)])
  (rkt.get-last-known-location))
```

No C++ code is generated for the above definitions, as they are both declared as `foreign`. For an example that does have a C++ translation, consider the following code, which introduces Magno-

lisp's predefined `predicate` type for boolean values, `var`iable declarations, `if` expressions and statements, and `do` and `return` constructs. The latter two are an example of language that does not directly map into Racket core language; the `do` expression contains a sequence of statements, with any executed `return` statement determining the value of the overall expression. Magnolisp syntax is not particularly concise, but shorthands can readily be defined, as is here demonstrated by the `declare-List-op` macro for declaring primitives that accept a `List` argument:

```
#lang magnolisp
; list and element data types (defined in C++)
(typedef List (#:annos foreign))
(typedef Elem (#:annos foreign))

(define-syntax-rule (declare-List-op [n t] ...)
  (begin (function (n lst)
           (#:annos [type (fn List t)] foreign))
         ...))

; list and element primitives (implemented in C++)
(declare-List-op [empty? predicate]
                 [head Elem]
                 [tail List])
(function (zero)
  (#:annos [type (fn Elem)] foreign))
(function (add x y)
  (#:annos [type (fn Elem Elem Elem)] foreign))

; sum of first two list elements
; (or fewer for shorter lists)
(function (sum-2 lst) (#:annos export)
  (if (empty? lst)
      (zero)
      (do (var h (head lst))
          (var t (tail lst))
          (if (empty? t)
              (return h)
              (return (add h (head t)))))))
```

The transcompiler-generated C++ implementation for the `sum-2` function is the following (but hand-formatted for readability). The translation is verbose, and could be simplified with additional optimizations; its redeeming quality is that it closely reflects the structure of the original code, which was made possible by our use of GCC-specific C++ language extensions (e.g., "statement expressions"):

```
MGL_API_FUNC Elem sum_2(List const& lst)
{
  return (is_empty(lst)) ? (zero()) :
    (({ __label__ b;
        Elem r;
        {
          Elem h = head(lst);
          {
            List t = tail(lst);
            if (is_empty(t))
              { r = h; goto b; }
            else
              { r = add(h, head(t));
                goto b; }
          }
        }
        b: r;
      }));
}
```

2.2 Magnolisp Implementation

The collection of techniques for embedding a transcompiled language within Racket, as described in this paper, only concern the front end of a transcompiler. Wildly differing designs for the rest of the compilation pipeline are possible; we merely sketch the structure of our Magnolisp-to-C++ compiler as a concrete example.

Magnolisp is implemented in Racket, and in a way there are two implementations of the language: one targeting the Racket VM, and one targeting C++. The `magnolisp` Racket language has the dual role of defining execution semantics for the Racket VM, and also effectively being the front end for the transcompiler.

Figure 1 shows an overview of the transcompiler architecture, including both the `magnolisp`-defined front end, and the `mglc`-driven middle and back ends. One detail omitted from the figure is that the macro-expanded `"a.rkt"` module gets compiled before it or any of its submodules are evaluated; if this is not done ahead of time, with the result serialized into a file as bytecode, it will get done on demand by Racket when the for-transcompile-time submodule is accessed.

Figure 2 illustrates the forms of data running through the compilation pipeline. The `"a.rkt"` module's transcompile-time code gets run when its `magnolisp-s2s` submodule gets *instantiated*, which means that variables are created for module-level definitions. Transcompilation triggers instantiation by invoking `dynamic-require` to fetch values for said variables (e.g., `def-lst`); the values describe `"a.rkt"`, and are already in the compiler's internal data format. Any referenced dependencies of `"a.rkt"` (e.g., `"num-types.rkt"`) 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 may be accessed via the `mglc` command-line tool, or programmatically via the underlying API. The expected input for these is a set of modules for transcompiling to C++. The compiler loads any transcompile-time code in the modules and their dependencies. Dependencies are determined by inspecting binding information for appearing identifiers, as resolved by Racket during macro expansion. Any modules with a `magnolisp-s2s` submodule are 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 in terms of term rewriting strategies. These are implemented based on a custom strategy combinator library inspired by Stratego (Bravenboer et al. 2008). The syntax trees prepared for the transcompilation phase use data types supporting the primitive strategy combinators that the combinator library expects.

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 `(TRUE)` or `(FALSE)`). The back end implements translation from Magnolisp core to C++ syntax (including e.g. lambda lifting), C++-compatible identifier renaming, splitting of code into sections (e.g.: public declarations, private declarations, and private implementations), and pretty printing.

3. Hosting a Transcompiled Language in Racket

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 the case of a transcompiled language, the expansion of the language's syntactic forms might produce 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.

In this section, we describe some of the details of that process for some transcompiled language L. Where the distinction matters, we use L_R to denote a language intended to also run in the Racket VM (possibly with mock implementations of some primitives), and

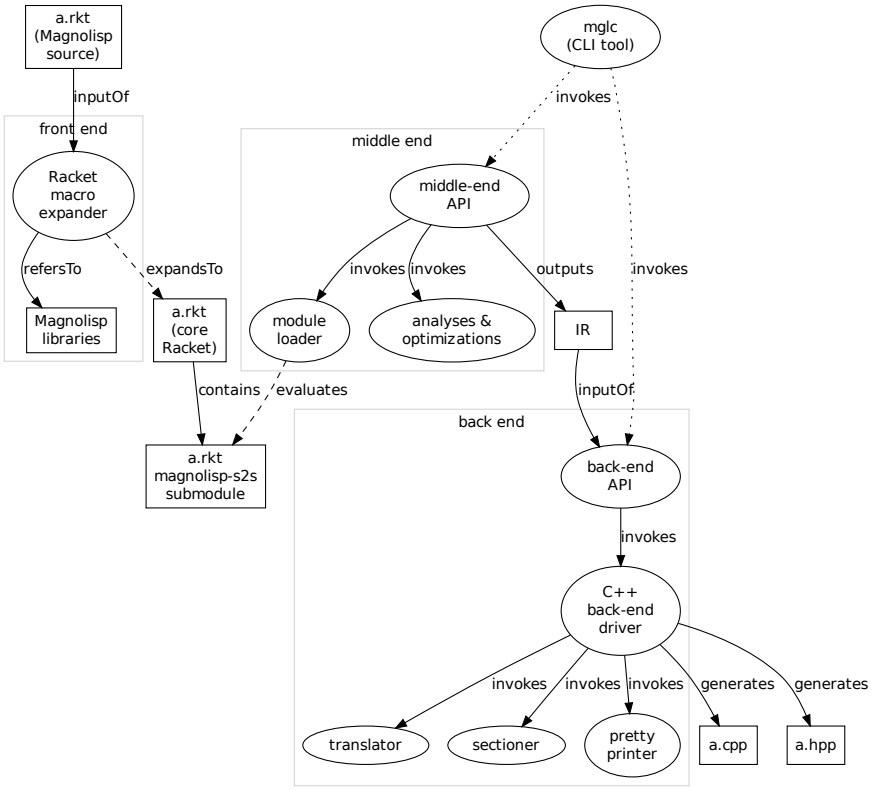


Figure 1: The overall architecture of the Magnolisp compiler, showing some of the components involved in compiling a Magnolisp source file `"a.rkt"` into a C++ implementation file `"a.cpp"` and a C++ header file `"a.hpp"`. The dotted arrows indicate that the use of the `mglc` command-line tool is optional; 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 ahead of time, in which case the module is already available in a macro-expanded form.

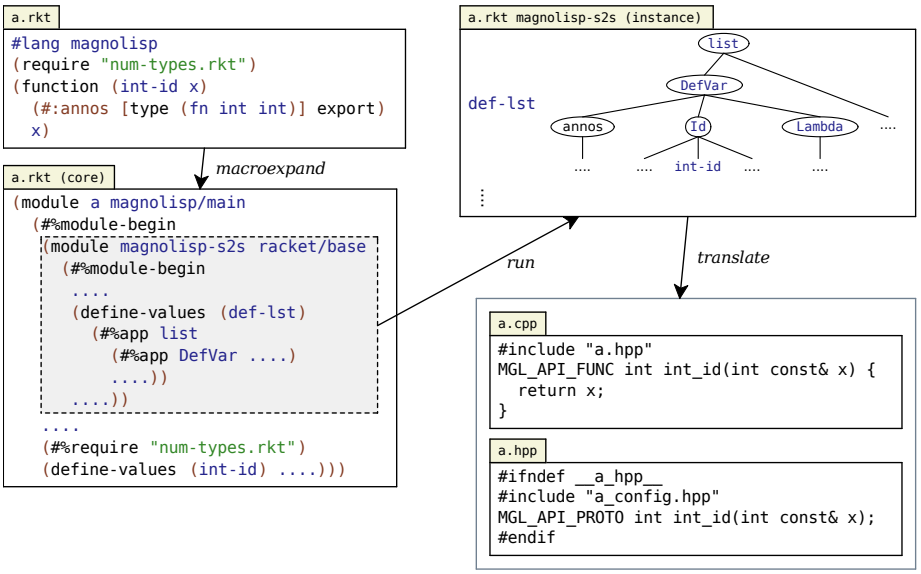


Figure 2: An illustration of the processing of a Magnolisp module as it passes through the compilation pipeline. Transcompile-time Racket code is shown in a dashed box.

L_C to denote a language that only runs through compilation into a different language.

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.[4] A language module is connected to the name L—so that it will be used by `#lang` L—by putting the module in a particular place in the filesystem or by appropriately registering the module's parent directory.

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 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 a language that uses 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 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.

For example, if `"main.rkt"` re-exports all of `racket`, then `#lang L` is just the same as `#lang racket`:

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

A language must at least export a macro named `#%module-begin`, because said macro 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 modules, 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.

For 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 ....)
```

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

The language definition facilities described so far are general, and useful regardless of whether the language is transcompiled or not. We now proceed to provide the specifics of this paper's transcompiled language implementation approach. In it, the `#%module-begin` macro in particular plays a key role, and overall, a Racket language L that is intended for transcompilation is defined as follows:

- Have the language's module export bindings that define the surface syntax of the language. The provided bindings should expand only to transcompiler-supported run-time forms. We describe this step further in section 3.2
- Where applicable, have macros record additional metadata that is required for transcompilation. We describe this step further in section 3.3
- Have the `#%module-begin` macro fully expand all the macros in the module body, so that the rest of the transcompiler pipeline need not implement macro expansion. We describe this step further in section 3.4
- After full macro expansion, have `#%module-begin` add externally loadable information about the expanded module into the module. We describe this step further in section 3.5
- Provide any run-time support for running programs 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 of L, we say that it is a *built-in*. As different execution targets may have different compilers, what is a built-in for one target may be a primitive for another. It is typically not useful to have built-ins for the Racket VM target, for which the `#%module-begin` macro may be considered to be the "compiler."

3.2 Defining Surface Syntax

To define the surface syntax of a language L, its implementation module's exports should ideally name a variable of L, a core language construct of L, or a macro that expands only to those constructs. Where the core language is a subset of Racket's, it should be ensured that only the transcompiler-supported set appears in an expansion. Where the core of L is a superset of Racket, the additional constructs should be encoded in terms of Racket's core forms.

Possible strategies for encoding foreign code forms include:

- **E1**. Use a variable binding to identify a core-language form. Use it in application position to allow other forms to appear within the application form. Subexpressions within the form can be delayed with suitable `lambda` wrappers, if necessary.
- **E2**. Attach information to a syntax object through its *syntax property* table; macros that manipulate syntax objects must propagate properties correctly.
- **E3**. Store information about a form in a compile-time table that is external to the module's syntax objects.
- **E4**. Use Racket core forms that are not in L (not under their original meaning), or combinations of forms involving such forms.

A caveat for **E2** and **E3** is that both syntax properties and compile-time tables are transient, and they generally become unavailable after a module is fully expanded, so any information to be preserved must be reflected as generated code in the expansion of the module; we describe this step further in section 3.5. Another caveat of such "out-of-band" storage locations is that where the stored data includes identifiers, one must beware of extracting identifiers out of syntax objects too early; if the identifier is in the scope of a binding form, then the binding form must be first expanded so that the identifier will include information about the binding.

In the case of L_R there are additional constraints to encoding foreign core forms, since 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 transcompilation-specific syntax in terms of Racket core forms which only appear in places where they do not affect Racket execution semantics.

Where the constraints **C1**–**C3** are troublesome, the fallback option is to have `#%module-begin` rewrite either the run-time code, transcompile-time code, or both, to make the program conform to expected core language. Such rewriting may still be constrained by the presence of binding forms, however.

The principal constraint on encoding a language's form is that a binding form in L should be encoded as a binding form in Racket, because bindings are significant to the process of hygienic macro expansion. Operations on a fully expanded module's syntax objects, furthermore, can reflect the accumulated binding information, so that a transcompiler may possibly avoid having to implement its own management of bindings. For the cases where a language's forms do not map neatly to a Racket binding construct, 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. Said 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` runtime 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)))
```

An example of adhering to constraint **C3** is the definition of Magnolisp's `typedef` form, for declaring an abstract type. 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 `#%magnolisp` has no useful definition in Racket, the evaluation of any `(#%magnolisp ....)` expressions is prevented. The `CORE` macro is a convenience for wrapping `(#%magnolisp ....)` expressions in an `(if #f .... #f)` form to "short-circuit" the overall expression so as to make it obvious to the Racket bytecode optimizer that the enclosed expression is never evaluated as Racket. The `let/annotate` form is a macro that stores the annotations `a ...`, which might e.g. include the name of `t`'s C++ definition.

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

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

Using a macro system for syntax definition offers several advantages compared to parsing in a more traditional way:[5]

- Where it is sufficient to define custom syntactic forms as macros, parsing is almost "for free." At the same time, the ability to customize a language's reader makes it possible for surface syntax not to be in Lisp's parenthesized prefix notation.
- Macros and the macro API provide a convenient implementation for "desugaring" and other rewriting-based program transformations. Such transformations can be written in a modular and composable way.
- For making L macro extensible, its implementation can simply expose a selection of relevant Racket constructs (directly or through macro adapters) to enable the inclusion of compile-time code within L modules.

3.3 Storing Metadata

We use the term *metadata* to mean data that describes a syntax object, but is not itself a core syntactic construct in the implemented language. Such data may encode information (e.g., optimization hints) that is meaningful to a transcompiler or other kinds of external tools. Some metadata may be collected automatically by the language infrastructure (e.g., source locations in Racket), some might be inferred by L's macros at expansion time, and some might be specified as explicit *annotations* in source code (e.g., the `export` annotation of Magnolisp functions, or the `weak` modifier of variables in the Vala language).

There is no major difference between encoding foreign syntax in terms of Racket core language, or encoding metadata; the strategies **E1**–**E4** apply for both. The main way in which metadata differs is that it does not tend to appear as a node of its own in a syntax tree. Any annotations in L do have surface syntax, but no core syntax, and hence they disappear during macro expansion; they do appear explicitly in unexpanded code, but such code cannot in general be directly analyzed, as unexpanded L code cannot be parsed. A more workable strategy is to have L's syntactic forms store any necessary metadata during macro expansion.

For metadata, storage in syntax properties is a typical choice. Typed Racket, for example, stores its type annotations in the a custom `'type-annotation` syntax property (Tobin-Hochstadt et al. 2011).

[5] A macro's process of validating and destructuring its input syntax can also be regarded as parsing, even though the input is syntax objects rather than raw program text or token streams (Culpepper 2012).

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. Magnolisp$_{it}$, a variant implementation of Magnolisp, uses an identifier table for metadata storage. Magnolisp$_{it}$ exports an `anno!` macro, which may be used to annotate an identifier, and is used internally e.g. by `function` and `typedef`. It is strictly a compile-time construct, and has no corresponding core syntax. Its advantage is that it may be used to post-facto annotate an already declared binding:

```
#lang magnolisp
(typedef int (#:annos foreign))
; MGL_API_FUNC int id(int const& x) { return x; }
(function (id x) x)
(anno! id [type (fn int int)] export)
```

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` is encoded as `(if #f (#%magnolisp 'anno ....) #f)` to prevent evaluation at Racket run time, and e.g. `[type ....]` expands to such a form, via the intermediate `CORE` form given in section 3.2:

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

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

The `let/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 are actually associated 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. Using the two features together is demonstrated by the following macro skeleton, which might be exported as the `#%module-begin` of a language:

```
(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))]))
```

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. The main problem with this strategy is that foreign binding forms would not be accounted for in Racket's binding resolution. That problem is compounded if foreign syntactic forms can include Racket syntax sub-forms; the sub-forms need to be expanded along with enclosing binding forms. To prevent these problems, a stop list is automatically expanded to include all Racket core forms if it includes any form so that partial expansion is constrained to the consistent case that stays outside of 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 expression to reconstruct the information 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-time versus run-time split, the information can be placed into a separate *submodule* within the module (Flatt 2013). 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 the enclosing module's bytecode. 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 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)))))]))
```

Wrapping `(module+ to-compile ...)` with `begin-for-syntax` makes the `to-compile` submodule reside at compilation time relative to the enclosing module, which means 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, be-

cause lexical-binding information is preserved in a syntax quote. Certain syntax-quoting forms—such as `quote-syntax/keep-srcloc`—additionally preserve source locations for syntax objects, so that a compiler can report errors or warnings in terms of a form's original source location.

Another natural representation choice is to use any custom intermediate representation (IR) of the compiler. Magnolisp, for example, processes each Racket syntax tree already within the module where macro expansion happens, turning them into its IR format, which also incorporates metadata. The IR uses Racket `struct` instances to represent abstract syntax tree (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`; it contains an expression that reconstructs the IR, albeit in a somewhat lossy way, excluding detail that is 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.

3.6 Run-Time Support

The modules implementing a Racket language may 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 the macro expansion of L can produce code referring to run-time variables, or when L exports bindings to run-time variables.

Any non-primitive run-time support variables are by definition defined in L itself, with each definition thus also compilable for the target. When L includes specific language for declaring primitives, then it may be convenient to define any variables corresponding to primitives in L, with any associated annotations; for L_R one would additionally specify any Racket VM implementation, either in Racket or another Racket-VM-hosted language. For variables representing built-ins of L_C, one might just use dummy initial value expressions, as the expressions are not evaluated, and the meaning of the variables is known to the compiler.

The primary constraint in implementing run-time support is that the Racket module system does not allow cyclic dependencies. Strictly speaking, then, any runtime library exported by a Racket module L *cannot* itself be implemented in L, but must use a smaller language. The `magnolisp` language, for example, exports the `magnolisp/prelude` module, which declares all the primitives of the language; the language of `magnolisp/prelude` is `magnolisp/base`, which does not include any runtime library.

The `magnolisp` language only exports four variables: the `#%magnolisp` built-in, and the `TRUE`, `FALSE`, and `predicate` primitives. The primitives are "semi-built-ins" in that the compiler knows that conditional expressions must always be of type `predicate`, and that the nullary operations `TRUE` and `FALSE` yield "true" and "false" values, respectively; this knowledge is useful during type checking and optimization:

```
#lang magnolisp/base

(require "surface.rkt")
(provide predicate TRUE FALSE)
(typedef predicate (#:annos [foreign
                             mgl_predicate]))
(function (TRUE) (#:annos [foreign mgl_TRUE]
                          [type (fn predicate)])
  #t)
(function (FALSE) (#:annos [foreign mgl_FALSE]
                           [type (fn predicate)])
  #f)
```

4. Evaluation

We believe that the presented Racket-hosted transcompilation approach is quite generic, in theory capable of accommodating a large class of languages. In practice, however, we would imagine it mostly being used to host *new* languages, with suitable design compromises made to achieve a high degree of reuse of the Racket infrastructure. Also, while macros are useful for language implementation alone, we would expect Racket's support for creating macro-extensible languages to be a significant motivation for choosing Racket as the implementation substrate.

Racket hosting should be particularly appropriate for research languages, as macros facilitate quick experimentation with language features, and design constraints should be acceptable if they do not compromise the researchers' ability to experiment with the concepts that are under investigation.

4.1 Language Design Constraints

The two design constraints for enabling effective Racket reuse that we have discovered are the following: (1) the hosted language's name resolution must be compatible with Racket's; and (2) S-expression-based syntax must be chosen to directly and effectively reuse Racket's default parsing machinery and existing macro programming APIs. The compilation requirement, in turn, may introduce constraints to the choice of core language, especially where one wants to output human-readable code.

Overloading as a language feature, for instance, appears a bad fit for Racket's name resolution. To alleviate the issue of naming clashes being more likely without overloading, Racket provides good support for renaming, including module system constructs such as `prefix-in` and `prefix-out` for mass renaming.

Defaulting to something S-expression based for surface syntax is advantageous, as then there is no custom reader to implement. Furthermore, as Racket syntax trees are also (enriched) S-expressions, and macros operate on them, one can then essentially use concrete syntax in patterns and templates for matching and generating code. This is comparable to the language-specific concrete syntax support in program transformation toolkits such as Rascal (Klint et al. 2009) and Spoofax (Kats and Visser 2010). Still, where important, other kinds of concrete syntaxes can be adopted for Racket languages, with or without support for expressing macro patterns in terms of concrete syntax; this has been demonstrated by implementations of Honu (Rafkind and Flatt 2012) and Python (Ramos and Leitão 2014), respectively.

In choice of core syntax, designing for natural and efficient mapping into the target language places fewer demands on the sophistication of the compiler's analyses and transformations. Magnolisp, for instance, is intended to map easily into most mainstream languages. It distinguishes between expressions and statements, for example, as do many mainstream languages (e.g., C++ and Java); making this distinction makes translation into said mainstream languages more direct.

4.2 Example Use Case: A Static Component System

As suggested above, macro-based extensibility might be an important motivation for implementing a Racket-based language, and

choosing a constrained core language might also be important for ease of transcompilation. One can reasonably wonder what the limits of macro-based expression then are, if constructs are defined in terms of their mapping into a limited run-time language. We address this question indirectly by considering a relatively advanced use case for macros as an example, namely that of component system implementation.

When organizing a collection of software building blocks, it can be useful to have a mechanism for "wiring up" and parameterizing said building blocks to form larger wholes (e.g., individual software products of a product line). Racket has a component system that includes such a mechanism; more specifically, the system supports *external linking*, i.e., parameterized reference to an arbitrary implementation of an interface (Owens and Flatt 2006). The system's *units* (Culpepper et al. 2005) are first-class, dynamically composed components.

Magnolisp lacks the run-time support for expressing units, and in this sense the language is severely constrained by its limited core language, and our lack of a comprehensive library of primitives for it. However, at compile time it has access to all of Racket, and hence enough power to implement a purely static component system. No such system is included, but to give an idea of how one might implement one, we provide a complete implementation of a rudimentary, yet potentially useful "component" system in figure 3.

Existing solutions suggest that it should also be possible to implement a more capable static component system in terms of Racket macros. Chez Scheme's `modules` support static, external linking, and have been shown to cater for a variety of use cases (Waddell and Dybvig 1999). Racket's built-in "packages" system (Flatt et al. 2012) resembles the Chez design, and is implemented in terms of macros, relying on features such as sub-form expansion, definition contexts, and compile-time binding. As packages are implemented statically, they require little from the run-time language.

```
#lang magnolisp
(define-syntax-rule (define<> x f e)
  (define-syntax f (cons #'x #'e)))

(define-syntax (use stx)
  (syntax-case stx (with as)
    [(_ f with new-x as fx)
     (let ([v (syntax-local-value #'f)])
       (with-syntax ([old-x (car v)] [e (cdr v)])
         #'(define fx
             (let-syntax ([old-x
               (make-rename-transformer #'new-x)])
             e))))]))

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

(function (->long x)
  (#:annos [type (fn int long)] foreign))

(define<> T id
  (let/annotate ([type (fn T T)])
    (lambda (x) x)))

; int int_id(int const& x) { return x; }
(use id with int as int-id)
; long long_id(long const& x) { return x; }
(use id with long as long-id)

; long run(int const& x)
;  { return long_id(to_long(int_id(x))); }
(function (run x) (#:annos export)
  (long-id (->long (int-id x))))
```

Figure 3: A primitive ''component'' system for Magnolisp. The macro `define<>` declares a named ''expression template'' `f`, and the macro `use` specializes such templates for a specific parameter `x`. Use of the two macros is demonstrated by a C++-inspired function template `id` with a type parameter `T`, also showing how macros can compensate for the lack of parametric polymorphism in Magnolisp. Corresponding `mglc`-generated C++ code is given in comments.

5. Related Work

While most languages previously implemented on Racket have been meant for execution only on the Racket virtual machine, a notable exception is Dracula (Eastlund 2012), which also compiles macro-expanded programs to ACL2. Dracula's compilation strategy follows the encoding strategy described in section 3.2 where syntactic forms expand to a subset of Racket's core forms, and applications of certain functions (such as `make-generic`) are recognized specially 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.

Sugar* (Erdweg and Rieger 2013) is a system for turning non-extensible languages into extensible ones. The resulting languages are extensible from within themselves, in a modular way, so that extensions are in scope following their respective module imports. While the aim of Sugar* is extended language into base language desugaring, one might also define a Sugar* "base language processor" that translates into another language before pretty printing. From among previously reported solutions, Sugar* perhaps comes closest to being a general solution to the implementation of transcompiled languages possessing the three characteristics listed in section 1.2. While Sugar* is liberal with respect to the definition of language grammars, one might arguably also gain guarantees of safe composition of language extensions through user-imposed discipline in defining them. The relative novelty of our solution is that it is itself based on a language-extension mechanism, whereas Sugar* is a special-purpose language implementation framework.

Silver (Wyk et al. 2010), like Racket, is a language capable of specifying extensible languages such that the extensions are modular and composable. Silver's specifications are based on attribute grammars (Knuth 1968), and the same formalism is used to specify both the base language and its extensions; therefore, even more so than with Racket, any extensions are indistinguishable from core language features. Silver supports safe composition of independently defined extensions by providing analyses to check whether extensions are suitably restricted to be guaranteed to compose; Racket provides some guarantees of safe composition, and even without analysis tools it tends to be obvious whether e.g. the "hygiene condition" (Kohlbecker et al. 1986) holds for the expansion of a given macro. While modular specification of syntax is supported by both Silver and Racket, only the former supports modular specification of semantic analyses. Such analyses—expressed as attribute grammar rules—may also be used to derive a translation to another language; Silver has been used to implement an extensible-C-to-C transcompiler, for example (Williams et al. 2014).

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 overloading 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 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.

Copilot (Pike et al. 2013) is also a Haskell-embedded language whose expressions are interpreted as AST constructions. Like Racket, Copilot has a core language, into which programs are transformed prior to execution. The Copilot implementation includes two alternative back ends for generating C source code; there is also an interpreter, which the authors have employed for testing. Copilot's intended domain is the implementation of programs to monitor the behavior of executing systems in order to detect and report anomalies. The monitoring is based on periodic sampling of values from C-language symbols of the monitored, co-linked program. Since such symbols are not available to the interpreter, the language comes built-in with a feature that the programmer may use to specify representative "interpreter values" for any declared external values (Pike et al. 2012); this is similar to Magnolisp's support for "mocking" of `foreign` functions.

The Terra programming language (DeVito et al. 2013) appears to take 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 lexical scope respecting way. Ultimately, however, Terra is not designed to support transcompilation, and compiles to binaries via Terra as a fixed core language. Another difference is Terra's emphasis on supporting code generation at run time, while ours 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 CGen core forms; 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.

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

Regardless of the implementation approach of a programming language, one might wish to extend it with additional features. Numerous motivating examples of language extensions are documented in literature (Hiraishi et al. 2007; Selgrad et al. 2014).

There are several technologies that specialize in language implementation (e.g., Rascal and Spoofax), and some of them (e.g., Silver) even focus on supporting the implementation and composition of independently defined language extensions. However, existing solutions generally lack specific support for the implementation of languages that are extensible from within themselves, and still aim to support convenient definition of extensions that compose in a safe manner. One exception is Racket, which supports the implementation of languages as libraries (Tobin-Hochstadt et al. 2011), and aims for safe composition of not only functions, but also syntactic forms. However, Racket-based languages have traditionally been run on the Racket VM, making Racket an unlikely choice for hosting transcompiled languages.

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, such that Racket's usual language definition facilities are not bypassed. Notably, the macro and module systems are still available, and may be exposed to the hosted language, to provide a way to implement and manage language extensions within the language. Furthermore, tools such as the DrRacket IDE still recognize the hosted language as a Racket one, are aware of the binding structure of programs written in it, and can usually trace the origins of macro-transformed code, for example.

Racket's macro system is expressive, allowing the syntax and semantics of a variety of language extensions to be specified in a robust way; general compile-time bindings for sharing of information between macros, for example, are supported. Scoping of language constructs can be controlled in a fine-grained manner using Racket's module system, and it is also possible to define or import macros for a local scope. With typical macros composing safely, and scoping control reducing the likelihood of macro naming clashes and allowing macros to be defined privately, pervasive use of syntactic abstraction becomes a real alternative to manual or tools-assisted writing of repetitive code.

The benefits of syntactic abstraction can furthermore be extended to any program-describing metadata, whether present to support transcompilation, or for other reasons; this can be done simply by having "data-as-code," thus making it subject to macro expansion.

Racket-hosted base language implementations can likewise leverage Racket's syntax manipulation facilities to perform macro-expansion-based transformations that produce non-Racket code. The approach indeed *requires* some macro-expansion time work to prepare separately loadable information for "transcompile time;" this does not preclude additional work performed in preparation for any optional Racket-VM-based run time.

Racket, with its general-purpose features and libraries, and ability to host program transformation domain specific sub-languages, may also be an attractive substrate for implementing the rest of a transcompilation pipeline.

Acknowledgements Carl Eastlund provided information about the implementation of Dracula. Magne Haveraaen, Anya Helene Bagge, and the SLE 2014 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

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.

Ryan Culpepper. Fortifying Macros. *J. Functional Programming* 22, pp. 439–476, 2012.

Ryan Culpepper and Matthias Felleisen. A Stepper for Scheme Macros. In *Proc. Wksp. Scheme and Functional Programming*, 2006.

Ryan Culpepper and Matthias Felleisen. Debugging Macros. In *Proc. Generative Programming and Component Engineering*, pp. 135–144, 2007.

Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic Abstraction in Component Interfaces. In *Proc. Generative Programming and Component Engineering*, pp. 373–388, 2005.

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.

Carl Eastlund and Matthias Felleisen. Hygienic Macros for ACL2. In *Proc. Sym. Trends in Functional Programming*, 2010.

Sebastian Erdweg and Felix Rieger. A Framework for Extensible Languages. In *Proc. Generative Programming and Component Engineering*, pp. 3–12, 2013.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *J. Functional Programming* 12(2), pp. 159–182, 2002.

Matthew Flatt. Composable and Compilable Macros: You Want it *When?* In *Proc. ACM Intl. Conf. Functional Programming*, pp. 72–83, 2002.

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

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 109–120, 2009.

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.

Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

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.

Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical System Theory* 2(2), pp. 127–145, 1968.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. Lisp and Functional Programming*, pp. 151–181, 1986.

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.

Scott Owens and Matthew Flatt. From Structures and Functors to Modules and Units. In *Proc. ACM Intl. Conf. Functional Programming*, 2006.

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience Report: A Do-It-Yourself High-Assurance Compiler. In *Proc. ACM Intl. Conf. Functional Programming*, 2012.

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring Embedded Systems. *Innovations in Systems and Software Engineering* 9(4), pp. 235–255, 2013.

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.

Oscar Waddell and R. Kent Dybvig. Extending the Scope of Syntactic Abstraction. In *Proc. ACM Sym. Principles of Programming Languages*, 1999.

Kevin Williams, Matt Le, Ted Kaminski, and Eric Van Wyk. A Compiler Extension for Parallel Matrix Programming. In *Proc. Intl. Conf. Parallel Processing (to appear)*, 2014.

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An Extensible Attribute Grammar System. *Science of Computer Programming* 75(1-2), pp. 39–54, 2010.

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