Research Outline: User-defined Core Forms

Language extension and DSL implementation in Racket

The essential mechanism for language-oriented programming in Racket is macro expansion. Macros define the meaning of new language forms by describing their translation to already-defined forms. The built-in forms that are not defined by such translation are called *core forms*.

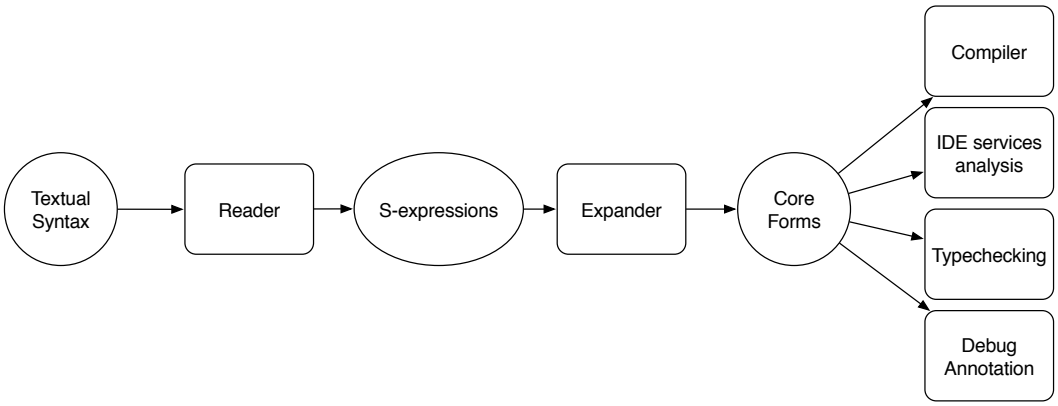

Figure 1

Figure 1 shows the language processing steps performed by the Racket implementation. After textual syntax is parsed into a concrete syntax tree of S-expressions by the reader, the expander traverses the syntax, applying macro transformers until all syntax parses as one of Racket's core forms. Programs in the language of core forms are understood by Racket's compiler.

Beyond the compiler, other analyses and tools operate on complete modules of core forms:

- Binding analysis and syntax highlighting for DrRacket (with mapping back via source locations)
- Compilation to alternate backends, like Pycket
- Errortrace annotation
- Typechecking in Typed Racket

Because user-defined language extensions and DSLs macro-expand to Racket's core forms, they inherit these generic services. In some cases extra annotations must be added by transformers so that analyses can present information to language users in terms of the surface syntax.

Other operations on programs such as bidirectional typechecking may be embedded within the expansion process itself, allowing custom implementations for each DSL. However, embedding within the expansion process is only straightforward for syntax-directed operations that are easy to divide up between the macros for individual language forms.

What of DSLs that need *domain specific* analysis, compilation, and tool support that cannot easily be embedded in the expansion process or inherited from Racket's generic tools?

User-defined core forms and applications

Racket should allow users to define their own core forms, which should behave in expansion as Racket's existing core forms do. This facility would allow processing of domain-specific intermediate representations in the same way that Racket's tooling processes the existing core forms language. Figure 2 shows the pipeline of language processing that user-defined core forms would enable.

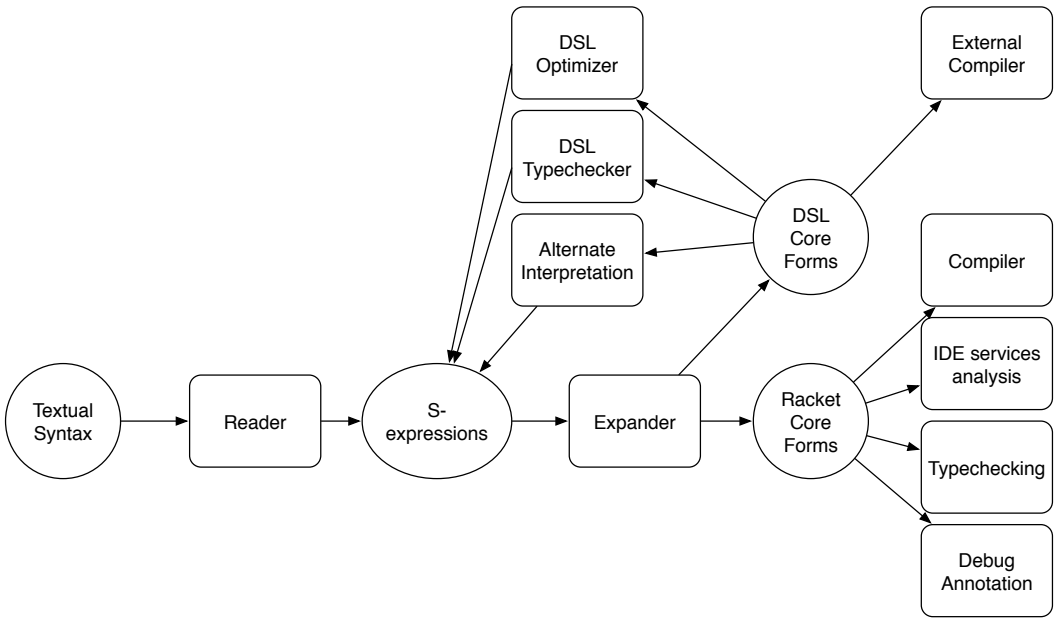

Figure 2

External back-ends

Some languages may want to use Racket's front-end infrastructure to support extensibility and integration with other languages by macro expansion, but use a different back-end for execution.

Examples:

- General-purpose GPU computation with CUDA, or graphics programming with shaders.
- Hackett, with compilation to GHC to take advantage of type information and optimize lazy evaluation.
- Verilog for programs that represent hardware descriptions, rather than computations.
- Redex, for language prototypes in PL research

Whole-module analyses

Other languages may ultimately be executed on Racket's back-end, but require processing that cannot easily be embedded in the expansion of individual language forms or be performed on Racket's core forms.

Examples:

- Typechecking that must process a whole module at a time, like Hindley-Milner type inference.
- Optimizations at the level of domain-specific language forms, but which require matching patterns of multiple forms or can only be done after some expansion to an intermediate language is complete.

Types

Just as we desire extensible languages of terms, we may also want extensible languages of types and extensible type-checkers. User-defined core forms would be particularly applicable for the representation of types for two reasons:

1. It does not make sense to expand types to core Racket terms, as there is not a sensible embedding; they represent different things. Furthermore, types may not even be present in a fully-expanded program, having been removed after static checking.

2. Extensions to a type system cannot typically be encoded by desugaring into a common language without losing important properties of the type system like decidable typechecking.

Multiple interpretations

Just as programs in Racket's core forms language are processed separately for compilation, IDE services, typechecking, and debugging instrumentation, it may be desirable to process a DSL program in a variety of ways.

A state machine description may be compiled to mutually recursive procedures in Racket, visualized with GraphViz, or verified with model checking. These different interpretations may be needed at different times, and it may be undesirable to embed all these interpretations into a single expanded program.

Existing support

Racket already has support for expansion processes that emulate some of the behavior of core forms. These techniques each fail to support the same kind of independent language extension that macros afford, require re-implementation of Racket's existing core-forms for each new language, or require brittle and unintuitive encoding.

Forms in the stop list and a custom expand loop

Macros that wish to expand some of the syntax they manipulate and process the corresponding core forms use the `local-expand` procedure to invoke the expander. This procedure takes a `stop-list` argument, specifying the names of language forms to leave un-expanded in the result. This mechanism provides limited support for user-defined core forms languages.

Such a macro must include code to parse the core forms revealed by `local-expand`,

perform scoping and binding operations, and invoke expansion on their subforms. For example, for a core-form language that includes Racket's core lambda and application, plus an additional application form `#%my-app`, the process might look like:

```
(define (custom-expand e ctx)
  (define e^ (local-expand e 'expression
                   (list #'#%my-app #'#%app #'#%plain-lambda)
                   ctx))
  (syntax-parse e^
    #:literals (#%my-app #%app #%plain-lambda)
    [(#%my-app e1 e2)
     #`(#%my-app #,(custom-expand e1 ctx)
                 #,(custom-expnad e2 ctx))]
    [(#%app f e ...) ; elided ]
    [(#%plain-lambda (x ...) b ...) ; elided ]
 ))
```

These operations mirror those that the expander performs for built-in core forms, but the existing implementations cannot be reused. If Racket's core forms are to be supported, the macro must reimplement the handling of those forms.

In order to provide the stop-list of core forms to `local-expand`, the complete set of possible such forms must be known. This makes it difficult to support processing of an extensible language of core forms, as is needed for example in an extensible typechecker.

Finally, the `stop-list` mechanism can be difficult to understand because it does not draw a conceptual distinction between forms that are designed to be core forms and those implemented by expansion. A macro could include the names of other macro-defined forms in the stop-list and parse and process them differently from their usual implementation.

Existing uses

This style of core form processing is widely used in existing languages in Racket both to work with the standard core form language and custom extensions. Examples include classes, units, `printing-module-begin`, `splicing-let`, and `#lang web-server`.

Stop Wrappers

It is also possible to define a protocol between macros that emulates core-form expansion:

```
(define-for-syntax (wrap e)
  #`(#%stop #,e))

(define-for-syntax (expand-unwrap e)
  (syntax-parse (local-expand e 'expression
                              (list #'stop))
    #:literals (#%stop)
    [(#%stop contents) #'contents]))

(define-syntax #%my-app
  (syntax-parser
    [(_ e1 e2)
     #:with e1^ (expand/unwrap #'e1)
     #:with e2^ (expand/unwrap #'e2)
     (wrap #'(#%my-app e1^ e2^))]))
```

Core forms are defined as macros that expand to a form with a wrapper that prevents further expansion when expanded by another macro that is aware of the convention.

The advantage of this protocol is that the language of cooperating core forms may be extended in just the same way that new macros are defined.

It would be possible to define counterparts of Racket's existing core forms in this style with similar duplication of the expander's implementation as in the previous strategy. However, these forms would have different identities than those built in to Racket, so macro definitions in existing languages would not expand to them. Programs could not mix forms from existing languages and new languages requiring new core forms.

Another encoding strategy with prefab structs as the wrapper could allow greater intermixture of user-defined and existing core forms, but does not support multiple expansion and results in a leaky abstraction.

Existing uses

Turnstile uses a variation on this strategy to delay expansion of type-erased code until after typechecking is complete.

Research Challenges

Operations on extensible languages

It is of little use to support extensibile core forms languages if there is no way to define extensible operations that process programs in those languages.

For example, Turnstile needs to perform several operations on types:

- substitution
- type equality
- subtyping
- kindchecking

The behavior of these operations needs to be defined for each new type represented as a core form.

Tool support

DrRacket analyzes Racket's core forms for services like syntax highlighting, binding arrows, and rename refactorings.

If core forms are only used as an intermediate language before expansion to Racket's core forms, the current technique of propagating information about source-level bindings and references using syntax properties is still workable, though awkward.

Another approach is needed for situations where language forms never expand to Racket's core forms, such as when an alternate back-end is used or forms represent types rather than programs.

Integration with the module system

Racket's module system helps manage dependencies and separate compilation. A DSL may want to re-use this infrastructure even while compiling to an alternate back-end.