Compiled, Extensible, Multi-language DSLs (Functional Pearl)

MICHAEL BALLANTYNE, Northeastern University, USA
MITCH GAMBURG, Unaffiliated, USA
JASON HEMANN, Seton Hall University, USA

Implementations of domain-specific languages should offer both extensibility and performance optimizations. With the new `syntax-spec` metalanguage in Racket, programmers can easily create DSL implementations that are both automatically macro-extensible and subject to conventional compiler optimizations. This pearl illustrates this approach through a new implementation of miniKanren, a widely used relational programming DSL. The miniKanren community has explored, in separate implementations, optimization techniques and a wide range of extensions. We demonstrate how our new miniKanren implementation with `syntax-spec` reconciles these features in a single implementation that comes with both an optimizing compiler and an extension mechanism. Furthermore, programmers using the new implementation benefit from the same seamless integration between Racket and miniKanren as in existing shallow embeddings.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Macro languages**; **Functional languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: DSL, miniKanren, logic programming, macros

ACM Reference Format:
Michael Ballantyne, Mitch Gamburg, and Jason Hemann. 2024. Compiled, Extensible, Multi-language DSLs (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP, Article 238 (August 2024), 24 pages. https://doi.org/10.1145/3674627

1 Reconciling the Benefits of Shallow and Deep Embeddings

When faced with a new problem domain, a functional programmer naturally thinks "this calls for a new language!" An internal domain-specific language (DSL) is the result. Two implementation strategies dominate the space of internal DSLs: shallow and deep embeddings [26]. These two embedding approaches have opposing tradeoffs. Shallow embeddings allow their users to extend the DSL and to fluidly interact with the host language. Deep embeddings come with a compiler that may perform optimizations. But, why shouldn't a DSL implementation come with all of these benefits? That is, we want:

(1) extensibility by DSL users,
(2) easy interaction between the DSL and the host language, and
(3) an optimizing compiler.

This pearl illustrates an architecture for DSLs that realizes all three properties. In this architecture, a macro expander transforms DSL programs written in an extensible surface language into a core language that is convenient for compilation. To add expressive power, this DSL core language is formulated as part of a multi-language [36] connecting the DSL to the host language. Boundary forms in the core language separate the host and DSL parts of the multi-language and introduce

Authors' Contact Information: Michael Ballantyne, ballantyne.m@northeastern.edu, Northeastern University, Boston, MA, USA; Mitch Gamburg, mitch.gamburg@gmail.com, Unaffiliated, Boston, MA, USA; Jason Hemann, jason.hemann@shu.edu, Seton Hall University, South Orange, NJ, USA.

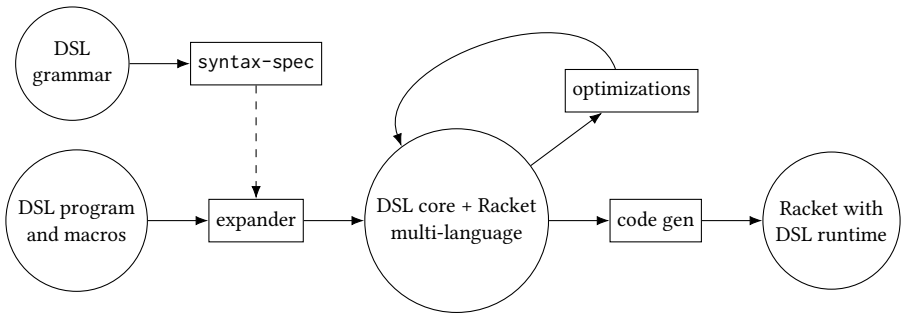

Fig. 1. The architecture for realizing compiled, extensible, multi-language DSLs.

static semantic and runtime checks to ensure interactions cannot break the DSL's invariants. A DSL compiler optimizes core language programs and generates efficient host language code. Ballantyne et al. [6] call DSLs implemented with this architecture *macro-extensible hosted DSLs*. Notice that the architecture is similar to that employed in many serious functional programming language implementations, including Clojure, Elixir, Haskell, Racket, and Scala. These languages all come with an extensible elaborator, a foreign function interface, and an optimizing compiler. Thus, the easy way to implement this architecture for DSLs is to extend such a host language's elaborator to work for DSL programs, too.

In Racket, the `syntax-spec` metalanguage [5] is the key ingredient to realize this architecture. Given a declarative specification of a DSL's syntax, `syntax-spec` extends Racket's hygienic macro system to the DSL. The DSL specification also describes the boundaries between the DSL and host language, establishing the multi-language structure mentioned above. The author of the DSL can then use conventional compiler technology to translate the DSL core language into Racket. Figure 1 outlines how the `syntax-spec` tool realizes the macro-extensible hosted DSL architecture. The dashed arrow indicates that `syntax-spec` generates the DSL expander.

We put the approach through its paces by implementing miniKanren, a relational programming DSL.[1] The miniKanren language is minimalist, limited to first-order relations, logical connectives, and constraints (Section 2). Declaring the syntax of miniKanren in `syntax-spec` is straightforward, and it yields a hygienic macro expander and multi-language structure (Section 3). Combining macros and host language interaction allows a user to extend miniKanren far beyond its core language (Section 4), to include relations defined by pattern matching and a database query interface connecting to SQLite. We connect the `syntax-spec`-based front-end to an optimizing back-end that realizes substantial performance gains (Section 5).

Along the way, we find that the combination of extensibility, multi-language interaction, and optimizing compilation yields more than the sum of its parts. The multi-language architecture makes it possible to define DSL extensions that look like a native part of the DSL but rely on the additional expressive power of the host (Section 4.3). And extensions benefit from optimizations, relieving macro authors from worrying about every detail of the code they generate (Section 5.2). Finally, the checks introduced by the multi-language boundary forms ensure the DSL compiler can soundly perform optimizations even in the presence of host-language code (Section 5.3).

[1]See miniKanren.org, and our implementation at github.com/michaelballantyne/hosted-minikanren

2 The miniKanren Language

The miniKanren DSL is a pure relational logic programming language, free of extra-logical effects. Its uses range from programming pedagogy [25] and medical research [24] to program synthesis research [9] and industrial applications [39]. The variant presented in *The Reasoned Schemer, Second Edition* [25] inspires our surface syntax and core language forms.

A miniKanren program consists of a set of relations and queries against those relations. The programmer poses a *query* about the possibility of some condition, in the context of the database of relations. A *goal* is the basic unit of computation. It can fail or succeed, and a goal can succeed multiple times. In miniKanren, as is usual with relational-logic programming languages, the basic goals are equations and other simple constraints. The programmer uses goal combinators to build up large goals from other goals—including calls to the defined relations.

The following example demonstrates how `run*` creates a query from a goal:

```
> (run* (destination) (direct 'SEA destination))
'(DEN BOS)
```

In this example a user asks to find all direct flights from Seattle and discovers there are exactly two: one to Denver and one to Boston. A `run*` expression consists of parameters with respect to which the programmer wants the answer printed and a goal the programmer wants to achieve. Here the goal is an invocation of the `direct` relation with two parameters: the quoted constant `'SEA` and the logical variable `destination`. Through the `run*` query interface, miniKanren delivers a list containing the values of `destination` for each of the two ways this query succeeds.

The `conde` form is the way that a goal can succeed in multiple alternative ways. The syntax of `conde` consists of a number of clauses. Each clause consists of some number of goals. Each clause represents the conjunction of its goals, and a `conde` form represents a disjunction of the clauses. Here is a query that uses the `conde` form:

```
> (run 3 (origin)
    (conde
      [(direct origin 'BOS)]
      [(direct origin 'HOU)]))
'(SEA DEN)
```

The `run 3` query asks for only the first three answers. We get two, because as it turns out the `direct` relation defines only one direct flight to Boston and one direct flight to Houston.

The DSL comes with another frequently-used goal combinator, `fresh`. The `fresh` form introduces auxiliary logic variables, scoped over its body; its body is some number of goals and represents their conjunction. Implementations of miniKanren usually track the accumulated constraints by threading some state through sequences of goals. The query below uses `fresh` to find the first layovers for all two-layover routes from Denver to San Francisco:

```
> (run* (lay1)
    (fresh (lay2)
      (direct 'DEN lay1) (direct lay1 lay2) (direct lay2 'SFO)))
'(HOU)
```

The answer `(HOU)` indicates that in the single way this query succeeds, the value of the logic variable `lay1` must be the city Houston. The answer returned by `run*` is always the list of values associated with the query variable in different solutions; the values of local variables such as `lay2` are not reported.

```
(defrel (route origin end path)
  (conde
    [(== origin end) (== path '())]
    [(fresh (hop remainder)
       (== path (cons (list origin hop) remainder))
       (absento origin remainder)
       (direct origin hop)
       (route hop end remainder))]))
```

```
(defrel (direct a b)
  (conde
    [(== a 'BOS) (== b 'SEA)]
    [(== a 'HOU) (== b 'SLC)]
    [(== a 'SEA) (== b 'DEN)]
    [(== a 'SEA) (== b 'BOS)]
    [(== a 'DEN) (== b 'HOU)]
    [(== a 'SLC) (== b 'SFO)]))
```

Fig. 2. The route and direct relations describing direct flights and routes in an airline network.

Searching this way for multi-hop routes is inelegant. Figure 2 shows the definition of direct together with a new relation route that relates origins, destinations, and paths between them. A relation definition consists of a name, the parameters, and a body goal. The goal states the condition under which the relationship holds. This body may refer to other relations; the set of relation definitions may be mutually recursive. The direct relation is a simple disjunction using the conde goal combinator. We write term equations with the == goal constructor.

The route relation is recursively defined. A route exists if the origin and end are the same and the path between them is empty. A route also exists when the path starts with a direct flight from origin to hop, and the rest of the path describes a route from hop to end. The absento form introduces a negated membership constraint, here used to forbid cyclic paths. The expression (absento origin remainder) means the term origin must be neither equal to, nor a subterm of, remainder.

Using route, we can query for all routes from Boston to Denver:

```
> (run* (path) (route 'BOS 'DEN path))
'(((BOS SEA) (SEA DEN)))
```

It turns out the only route has a layover in Seattle.

2.1 A Better Way

With the flights example in hand, we can illustrate what we really want to achieve. First, conde is too primitive for functional programmers' taste; we'd rather use pattern matching. Second, there are many more flights in the USA than we can enumerate in direct. Consulting a database instead would allow us work with real airline data. Finally, we want all of this to run as fast as it can, thanks to an optimizing compiler.

```
(defrel (route-m origin end path)
  (matche (origin end path)
    [(a a '())]
    [(a b (cons (list a layover) remainder))
     (absento a remainder)
     (direct-db a layover)
     (route-m layover b remainder)]))
```

```
(define-facts-table flights [from to]
  #:initial-data (download-flights-csv))

(defrel (direct-db a b)
  (query-facts flights a b))
```

Fig. 3. Revisions of the direct and route relations using database and pattern-matching extensions.

By applying our DSL architecture to miniKanren, we can achieve all these goals. In particular, pattern matching and database connectivity become user-definable extensions. Figure 3 displays a revision of the flights program using such extensions. The revision of route, called route-m, uses the matche pattern matching form in place of the more verbose combination of conde, fresh, and

== used in the original program. On the right side of the figure, the `define-facts-table` form creates a new SQLite database and initializes it with data from a CSV file, downloaded from the internet. Consulting this database requires one more form, `query-facts`. It represents a miniKanren goal that succeeds with different assignments of the variables `a` and `b` for each row resulting from an SQLite query. Both `define-facts-table` and `query-facts` are once again user-defined extensions, but must interact with the host language in order to access the external world.

The essential step to safely mixing these extensions with optimizing compilation is to define an appropriate target language for expansion. This core language should integrate host-language code without breaking miniKanren's invariants and should be sufficiently high level to support domain-specific optimizations. Let's proceed to define such a core language with `syntax-spec`.

3 Defining miniKanren with syntax-spec

Implementing miniKanren via `syntax-spec` requires three steps. First, write out a conventional grammar for the DSL, annotated with a specification of miniKanren's binding structure. Second, define the boundaries with the host language, which also serve as the entry points to the DSL compiler. Finally, we throw some syntactic sugar into the mix.

3.1 The Core Language

Following the informal presentation in the preceding section, Figure 4 shows the grammar of the core miniKanren language in both EBNF style and in `syntax-spec`. These grammars have an unstated closure condition: language forms not mentioned—specifically host-language Racket forms—are statically disallowed in miniKanren term and goal positions.

```
                                        (binding-class term-variable)
                                        (binding-class rel-name)

<quoted> :=                             (nonterminal quoted
          | <number>                      n:number
          | <id>                          s:id
          | ()                            ())

<term>   :=                             (nonterminal term
          | <id>                          x:term-variable
          | (quote <quoted>)              (quote t:quoted)
          | (cons <term> <term>)          (cons t1:term t2:term))

<goal>   :=                             (nonterminal goal
          | succeed                       succeed
          | fail                          fail
          | (== <term> <term>)            (== t1:term t2:term)
          | (absento <term> <term>)       (absento t1:term t2:term)
          | (disj <goal> ...+)            (disj g:goal ...+)
          | (conj <goal> ...+)            (conj g:goal ...+)
          | (fresh1 (<id> ...) <goal>)    (fresh1 (x:term-variable ...) b:goal)
                                          #:binding (scope (bind x) b)
          | (<id> <term> ...+)            (r:rel-name t:term ...+))
```

Fig. 4. An EBNF-style grammar for miniKanren (left) and the corresponding `syntax-spec` version (right). The `syntax-spec` definition also includes binding structure information (underlined).

The core language is more minimalist than the surface syntax used in Section 2. It provides a simple disjunction form `disj` instead of `conde`. The `fresh1` form is like `fresh`, but with only one body goal. Core language programs make conjunction explicit with `conj` instead of relying on the implicit conjunction provided by the surface-syntax `conde` and `fresh`. The language also includes the primitive goals `succeed` and `fail`, which do what their names suggest. Like function applications in any Lisp, relation applications simply begin with the relation name.

The language of terms consists of term variable references, quoted data, and `cons` pairs. The syntax of quoted data is a subset of that found in Racket. It includes numbers, identifiers acting as symbolic data, and the empty list written as `()`.

Besides the grammar, a `syntax-spec` definition requires a little bit of the static semantic specification. The DSL implementer uses the `#:binding` keyword to describe the binding structure. In `fresh1`, the binding rule indicates that the name `x` is bound in a new scope and is visible in the body goal `b`. Unlike in the EBNF version, the `syntax-spec` grammar requires a user to name each form's sub-forms (for instance, `x` in `x:term-variable`). These names are required precisely so that sub-forms can be uniquely identified in binding specifications.

Above the grammars are the `binding-class` definitions. These forms define the set of term variables and the set of relation names. We use these classes to make the syntax specification precise: a reference to a name is only valid if it has the same class as the binding to which it resolves. We underline those portions of the miniKanren syntax specification that relate to binding structure in order to highlight them.

3.2 The Boundary with Racket

The critical points of any multi-language design are the boundaries between the host language and the DSL. The `defrel` and `run` forms are the main entries into miniKanren from the Racket language grammar. Both are extensions to the *Racket* language; as such, they are special because they permit miniKanren sub-expressions. To make this mixing of syntax semantically safe, our multi-language implementation must, at runtime, check and protect the values that flow from miniKanren to Racket and vice versa. We draw inspiration from the work of Matthews and Findler [36] on multi-language formalization. Specifically, we borrow the idea of explicit syntactic boundaries with corresponding value translations or contracts [21].

As shown in Figure 5, the DSL designer using `syntax-spec` writes `host-interface` forms to add DSL entry points. Since the `run` form is an extension to the class of expressions in the Racket language, we use `host-interface/expression` to introduce this new syntax. The pieces of the definition are a specification of the new syntax, a binding rule, and an invocation of the DSL compiler. A DSL designer writes the syntax specification and binding rule syntaxes the same way as in the grammar in Figure 4. The `racket-expr` annotation means that a Racket expression is expected for the quantity argument `n`. A `run` compiles to a Racket expression via `compile-run`.

```
(host-interface/expression (run n:racket-expr (q:term-variable) g:goal)
  #:binding (scope (bind q) g)
  (compile-run #'n #'q #'g))

(host-interface/definition (defrel (r:rel-name x:term-variable ...+) g:goal)
  #:binding [(export r) (scope (bind x) g)]
  #:lhs [#'r]
  #:rhs [(compile-relation #'(x ...) #'g)])
```

Fig. 5. Host interface forms creating the boundary between Racket and miniKanren.

This expression performs the search for `n` results and returns them as a list. To make this safe and report reasonable errors, the generated code inserts a contract check to enforce that the expression `n` evaluates to a natural number.

Unlike with `run`, a `defrel` form is only permitted in Racket definition contexts. Thus the definition of `defrel` uses `host-interface/definition`. A `defrel` exports the relation's name into the surrounding definition context, and it also scopes all the relation's parameters over the body. That is, the `#:binding` keyword can introduce not only locally scoped identifiers, but also identifiers scoped in the surrounding context, using `export`. Each `defrel` compiles to a Racket function definition of the shape `(define lhs rhs)`. For the underlying Racket definition we want to use a name derived from the relation name. The compiler code introduced by `#:lhs` produces that name, and the code introduced by `#:rhs` produces said Racket function.

3.3 Elaboration

DSL creators want both a small core language to compile and a larger, convenient-to-use surface language. This dual goal is traditionally achieved with an elaboration pass at the front end of the compiler. The Racket language itself performs elaboration by defining the surface language with macros that are expanded by the Racket macro expander into the Racket kernel language.

The `syntax-spec` system makes this part easy. It generates a bespoke hygienic expander from the declarative language specification with just a few hints from the DSL designer. Concretely, the DSL designer defines classes of extensions and indicates in the grammar where each may be used. Then, the designer writes desugarings as conventional macros.

For miniKanren, we define two extension classes called `term-macro` and `goal-macro`:

```
(extension-class term-macro)
(extension-class goal-macro)
```

The different extension classes enable the DSL-specific expander to offer users precise error messages. The DSL designer uses the `#:allow-extension` keyword in `nonterminal` definitions to indicate the positions in the grammar that permit syntax sugar. For miniKanren we allow term macros in the `term` nonterminal and goal macros in the `goal` nonterminal by modifying the declarations of Figure 4:

```
(nonterminal term #:allow-extension term-macro
  #| elided |#)
(nonterminal goal #:allow-extension goal-macro
  #| elided |#)
```

The `define-dsl-syntax` form installs a desugaring. Here we desugar the multi-body `fresh`:

```
(define-dsl-syntax fresh goal-macro
  (syntax-parser
    [(fresh (x:id ...+) g ...+)
     #'(fresh1 (x ...) (conj g ...))]))
```

An extension definition requires a name for the macro (`fresh`), the extension class to which it belongs (goal macros), and the rewriting rule. We define the rewriting using Racket's standard `syntax/parse` library [15, 16]. The `syntax-parser` form defines a function that pattern matches on the syntax it receives. The syntax template introduced by the `#'` characters constructs the expansion, filling in the values of pattern variables from the pattern match.

The term language of Figure 4 permits quoted literal constants and `cons`-based lists, but it does not directly include a Racket-like `list` function. Without intervention, this would demand the programmer quote each individual constant symbol and use binary `cons` to construct all complex terms. Our miniKanren implementation provides `list` as a term macro. We omit the implementation of `list`, but provide an example of the desugaring, which is straightforward:

```
(== (list (list 'a)) (list x))
;; desugars to
(== (cons (cons 'a '()) '()) (cons x '()))
```

Racket's module system supports controlling the visibility of not just function and value definitions, but also syntax definitions [22]. We now have the surface forms and the core forms, but we want to present only the nice surface language to the DSL user. Because we are in Racket, we can use the module system to expose only those forms intended to be part of the surface. For instance, the miniKanren implementation module exports `fresh` but not `fresh1`:

```
(provide (all-except-out fresh1 conj disj))
```

3.4 Runtime Semantics

Our miniKanren implementation matches the semantics of the popular `faster-miniKanren` implementation [4], aiming to serve as a drop-in replacement. The `faster-miniKanren` system is a shallow embedding, so it acts as a useful point of comparison for our alternative implementation strategy. Our compiler also re-uses the `faster-miniKanren` run-time system's implementation of components such as unification and miniKanren's signature interleaving search [32]. Thus the differences between the two systems are in the `syntax-spec`-based front-end and the optimizations introduced by our back-end compiler. We discuss those optimizations in detail in Section 5.

3.5 The Big Picture

Having seen the `syntax-spec` language specification for miniKanren, we can fill in some of the details from the architecture diagram of Figure 1. The grammar, binding specifications, and extensions described in Sections 3.1–3 fully characterize how parsing and desugaring should behave. The `syntax-spec` system uses this information to generate the compiler's front end. The expander eliminates all DSL surface-syntax sugar. The DSL implementer can therefore write a compiler back end that handles just the core language.

The DSL-host boundary forms (such as `run`) call functions that act as entry points to the back end of the DSL compiler (such as `compile-run`). The back end is written as Racket code that runs at compile time and can consist of as many passes as needed. The compiler writer can use the entire Racket language to structure the compiler including metalanguages such as `syntax-parse`. The back-end passes implement the right half of Figure 1: optimizations and code generation. The emitted code relies on the runtime system from `faster-miniKanren`.

4 Extension and Mixing Like a Shallow Embedding

Using `syntax-spec` imbues our miniKanren implementation with the same powers of extension and intermixing of DSL and host-language code one gets with a shallow embedding. Specifically, miniKanren programmers can extend the syntax of the DSL with macros, and they can commingle host and DSL code in a disciplined manner with boundary forms.

Programmers working in shallow embeddings of miniKanren have dreamed up many language extensions and developed applications that take advantage of the ability to intermix miniKanren and Racket. Together these programs make the miniKanren ecosystem a kind of natural experimental environment.

```
(define-dsl-syntax matche goal-macro
  (syntax-parser
    [(matche (arg ...+) [pats g ...] ...+)
     #:with ([pats^ xs] ...) (map compile-pats (attribute pats))
     #'(fresh (ls)
         (== ls (list arg ...))
         (conde [(fresh xs (== pats^ ls) g ...)] ...))]))

;; (Listof Pattern) -> (Pair Term (Listof TermVar))
(define (compile-pats pats) #| elided |# )
```

Fig. 6. The implementation of the matche pattern matching extension.

In this section, we reproduce some of these examples. By doing so, we show how syntax-spec endows an optimizing implementation with the same freedom of language extension and intermixing as a shallow embedding—with little effort.

4.1 Extensibility

Many of those programmer-designed extensions to shallow miniKanren embeddings are simple host-language macros. Hence it is perhaps unsurprising that we can similarly implement them as DSL macros. We already explained how to do it: the DSL programmer uses the same extension mechanism as the DSL designer does for syntax sugar. The only difference between an end-programmer language extension and built-in syntactic sugar is who designs it. Since DSL extensions are macros that expand to DSL core-language code, the compiler's expectations about the source code continue to hold.

Concretely, some shallow miniKanren implementations include a matche [29] pattern-matching form, like that used in route-m of Figure 3. The first form in each matche clause is a pattern. The process of pattern matching introduces the necessary logic variables and unifies a term corresponding to the pattern against the relation parameters.

The matche implementation is a fairly pedestrian goal-macro, which we show in Figure 6. The implementation compiles each pattern group to a term expression pats^ and a list of pattern variables xs. This process relies on the compile time helper function compile-pats, whose definition we omit. The macro then constructs a goal that introduces a name for the arguments list followed by a conde. Each conde clause introduces the pattern variables with fresh, unifies the term expression pats^ with the arguments list, and executes the sequence of goals g ... as a conjunction.

Since the host and DSL use the same extension system, we can even write macros that generate mixed-language code. For instance, a programmer might want to express a relation definition like route-m even more concisely. In route-m, we list the parameters in the header, and then match against that same parameter list. We can abstract over that duplicated syntax with defrel/matche,

```
(define-syntax defrel/matche
  (syntax-parser
    [(defrel/matche (name:id arg:id ...+)
       clause ...+)
     #'(defrel (name arg ...)
         (matche (arg ...) clause ...))]))
```

```
(defrel/matche (route-m origin end path)
  [(a a '())]
  [(a b (cons (list a layover) remainder))
   (absento a remainder)
   (direct a layover)
   (route-m layover b remainder)])
```

Fig. 7. The defrel/matche macro and an even more concise re-definition of the route relation.

an ordinary Racket macro that generates a `defrel` with a `matche` as its body. The left-hand side of Figure 7 shows the definition, and the right-hand side contains a re-implementation of `route-m` using this new abstraction.

4.2 Mixing DSL and Host-Language Code

Sometimes miniKanren programmers want just a touch of Racket *inside* their miniKanren programs. The miniKanren language is a distillation of purely relational constraint logic programming that deliberately eschews features such as side effects and higher-order control flow. However, sometimes you just need to do a little `printf` debugging.

```
> (define (succeed/print str)
    (printf str)
    (expression-from-goal succeed))
> (run 1 (q)
    (fresh (x)
      (== q (list x 'cat))
      (goal-from-expression
        (let ([str (format "value of ~a: ~a\n" 'q (expression-from-term q))])
          (succeed/print str)))))
value of q: (#<mk-lvar> cat)
'((_.0 cat))
```

Fig. 8. Using Racket to construct and print a string through the host FFI.

Figure 8 demonstrates how a programmer can write ordinary Racket code within a miniKanren program. This interaction requires extending miniKanren with some additional language-boundary forms. The query contains a use of the `goal-from-expression` boundary form, which allows a miniKanren programmer to include some Racket code inside a goal. In this example, the Racket code constructs a string containing information about the variable `q` and its value. This string construction relies on yet another cross-language boundary to access a miniKanren term variable from the Racket context. The `expression-from-term` form admits a miniKanren term in a Racket context; the cross-language translation of the value is trivial except for fresh miniKanren logic variables, which are opaque to the Racket context. The final line in the `goal-from-expression` form contains an ordinary Racket function call to `succeed/print`, which consumes and prints a string.

Beyond allowing Racket code to access term variables and perform side effects in the context of a goal, `goal-from-expression` also allows Racket code to define the behavior of the goal—whether it succeeds or fails, and whether it constrains any variables. The `succeed/print` body returns a goal value created by the `expression-from-goal` boundary form to represent this behavior. In this example the goal is simply `succeed`—Section 4.3 shows a more sophisticated use. An additional boundary form `term-from-expression` form allows Racket code to compute a term; it is not used in this example.

Adding these features requires just four additions to the grammar of Figure 4 and the `syntax-spec` definition of miniKanren. Figure 9 shows these additions, again with an EBNF on the left and the actual `syntax-spec` notation on the right. The boundaries that allow Racket code in miniKanren contexts are defined as productions in the `term` and `goal` nonterminals that have `racket-expr` subexpression positions. The `expression-from-goal` and `expression-from-term` boundary forms extend the class of Racket expressions. Thus, we implement them with `host-interface/expression` just like the `run` form.

```
<term> :=
  ....
  | (term-from-expression <racket-expr>)

<goal> :=
  ....
  | (goal-from-expression <racket-expr>)

<racket-expr> :=
  ....
  | (expression-from-term <term>)

<racket-expr> :=
  ....
  | (expression-from-goal <goal>)
```

```
(nonterminal term
  ....
  (term-from-expression e:racket-expr))

(nonterminal goal
  ....
  (goal-from-expression e:racket-expr))

(host-interface/expression
  (expression-from-term t:term)
  (compile-expression-from-term #'t))

(host-interface/expression
  (expression-from-goal g:goal)
  (compile-expression-from-goal #'g))
```

Fig. 9. Extension to the grammar of Figure 4 with additional multi-language boundary forms.

As mentioned in Section 3.2, connecting the parts of a multi-language safely involves inserting value translations or contracts at the boundaries. Matthews and Findler [36] describe two ways of relating the kinds of values found in the connected languages. For goals, we choose a lump embedding: the boundary forms seal miniKanren goals as opaque lumps. The only action Racket code can perform with these values is to return them to miniKanren. For terms, we choose the natural embedding: miniKanren terms such as lists and numbers translate to the equivalent data structures in Racket, while logic variables remain opaque. Many Racket values such as vectors and structures have no translation to miniKanren term values, so passing such values from Racket to miniKanren results in a contract error.

To show how these translations are inserted, Figure 10 illustrates the compilation of the example in Figure 8. The compilation of each cross-language boundary inserts a call to a value translation function. Together, `seal-goal` and `unseal-goal` implement the lump embedding for goal values. The `translate-term` operation is responsible for implementing the natural embedding of term values, translating term data to Racket values while sealing logic variables. This Racket-safe version of a miniKanren term can find its way back to a miniKanren context through `term-from-expression`, which unseals the logic variables.

Some of the boundary forms need to access information from the miniKanren state. In our runtime, the current value of a logic variable depends on the received state, which contains information on the history of unifications during the program's execution. We implement goals as functions from a state to a stream of states representing the nondeterministic result of the goal. In Figure 10, the `goal-from-expression` form compiles to such a function, making the `state` variable accessible to the generated code within. The compilation of `expression-from-term` uses this state with `substitute` to fill in known logic variable values before passing the term to Racket. The compilation of `goal-from-expression` uses the state via `apply-goal` to continue execution with the goal produced by the Racket code.

```
(goal-from-expression
  (let ([str (format "~a: ~a\n" 'q (expression-from-term q))])
    (printf str)
    (expression-from-goal succeed)))

;; -Compiles to->

(lambda (state)
  (let ([goal-val
         (let ([str (format "~a: ~a\n" 'q (translate-term (substitute q-rt state)))])
           (printf str)
           (seal-goal succeed-rt))])
    (apply-goal (unseal-goal goal-val) state)))
```

Fig. 10. The above `goal-from-expression` sub-form from the example in Figure 8 compiles to the below Racket implementation code. We have in-lined here the body of `succeed`/`print` in both examples for clarity.

Convenience. Forcing the programmer to wrap every reference to a miniKanren variable within Racket with `expression-from-term` would be ergonomically awkward. Our implementation co-operates with the Racket expander to automatically wrap `expression-from-term` around every term variable reference within a Racket context. This convenience feature means the miniKanren programmer pays no additional price for precisely delineated and checked language boundaries.

The implementation of this convenience feature relies on tools provided by `syntax-spec`. The various compiler entry points eventually invoke the `compile-goal` function shown in Figure 11, which is responsible for running the optimizer and generating Racket code for the goal.

```
(define (compile-goal g)
  (define g^ (optimize-goal g))
  #`(with-reference-compilers ([term-variable compile-expression-from-term])
      #,(generate-goal g^)))
```

Fig. 11. The compile-time function `compile-goal`.

In order to implement the automatic wrapping of term variables, part of the code `compile-goal` generates is actually an annotation for the Racket expander as extended by `syntax-spec`. The `with-reference-compilers` form specifies how to transform DSL-language variable references within its body. The reference to the `term-variable` binding class from the `syntax-spec` definition of miniKanren indicates that all term variable references should be transformed. We provide the function `compile-expression-from-term` as the reference compiler. This is the same function used as the entry point to compile `expression-from-term` boundary forms. All references that appear in the Racket code produced by `generate-goal` are transformed accordingly.

4.3 Host Code in DSL Extensions

The synthesis of extensibility and host language interoperation produces some surprisingly powerful behaviors. For instance, the PL enthusiast will quickly tire of writing and rewriting that same kind of `printf` logic from Section 4.2. Instead, the informed enthusiast will instinctively reach for the means to abstract over syntactic boilerplate: macros.

```
(define-syntax trace-defrel
  (syntax-parser
    [(_ (name:id a*:id ...) b:goal/c)
     #'(defrel (name a* ...)
         (goal-from-expression
           (begin (printf "~s\n" (list 'name a* ...))
                  (expression-from-goal b))))]))
```

Fig. 12. The `trace-defrel` form combining extension with host FFI interaction

For example, the `trace-defrel` macro in Figure 12 provides a quick way for the programmer to see the values of a relation's arguments at every entry to that relation. Users of `trace-defrel` need not even be aware that it uses cross-language code, because it is all is hidden behind the abstraction. This particular macro is defined with Racket's standard `define-syntax` rather than `syntax-spec`'s `define-dsl-syntax` because relation definitions sit at the boundary with Racket.

As previewed in Section 2.1, database access provides a more substantial application of the combination of syntactic extension and host interoperation. Recall that Figure 3 re-implements the `direct` flights relation using an extension that connects miniKanren to an SQLite database. We now have all the pieces we need to define this extension. The `define-facts-table` form is a standard Racket macro. It expands to code that uses the Racket `database` library to create and populate an SQLite database table; we elide its implementation. More interesting is the `query-facts` goal macro defined in Figure 13, which straddles the boundary between Racket and miniKanren.

```
(define-dsl-syntax query-facts goal-macro
  (syntax-parser
    [(_ table term ...)
     #'(goal-from-expression
         (query-facts-rt table (list (expression-from-term term) ...)))]))
```

Fig. 13. The `query-facts` goal macro combining extension with host FFI interaction.

The `query-facts` macro consumes a reference to a facts table and a sequence of miniKanren term expressions that should evaluate to either atomic miniKanren values or logic variables. The macro expands to a goal formed from a Racket expression that will, at runtime, actually execute a database query. The expansion leverages the `expression-from-term` multi-language boundary form to check that the argument syntax `term` is valid term syntax and to convert the term value to a Racket value for use in the runtime helper.

The implementation of the runtime support for the extension is presented in Figure 14. The entry point `query-facts-rt` relies on three helper functions. First, it uses the `wildcardify` function to transform sealed logic variables into a "select all" wildcard that the database understands. Then, it executes the query with `do-query` to produce a list of matching table rows. Finally, it uses `unify-query-results` to non-deterministically unify the original term arguments with each possible option returned from the database. The `wildcardify` and `do-query` implementations are straightforward Racket functions using parts of the miniKanren runtime and Racket database library. The `unify-query-results` function has the most interesting multi-language interaction.

The `unify-query-results` function takes an arbitrarily long list of matching rows from the database lookup and produces a goal that non-deterministically unifies `args` with each of these values in turn. These unifications can fail because of delayed constraints (like `absento` from Figure 2) that cannot map directly to restrictions in the query. When the list of results is non-empty,

```
;; Table (Listof TermVal) -> GoalVal
(define (query-facts-rt table terms)
  (define matching-rows (do-query table (map wildcardify terms)))
  (unify-query-results matching-rows terms))

;; TermVal -> (Or Atom Wildcard)
;; THROWS when term is instantiated to a non-atom
(define (wildcardify term) #| elided |# )

;; Table (Listof (Or Atom Wildcard)) -> (Listof (Listof Atom))
(define (do-query table args) #| elided |# )

;; (Listof (Listof Atom)) (Listof TermVal) -> GoalVal
(define (unify-query-results query-res args)
  (match query-res
    ['() (expression-from-goal fail)]
    [(cons fst rst)
     (expression-from-goal
       (conde
         [(== fst args)]
         [(goal-from-expression (unify-query-results rst args))]))]))
```

Fig. 14. The runtime portion of the `query-facts` extension.

the function returns a `conde` goal to implement that nondeterministic choice. In the second disjunct we make the recursive call to `unify-query-results`. This mixing of recursive Racket computation with goal construction relies on nested language boundaries.

To the miniKanren programmer using it in Figure 3, the `query-facts` extension looks like any other miniKanren form. Creating a DSL extension lets us hide the implementation details of complex cross-language operations behind simple, familiar looking syntax.

5 Optimizing Like a Deep Embedding

Like in a deep embedding, our miniKanren compiler has access to a syntactic representation of DSL program fragments, and it can thus realize all kinds of optimizations. It uses a traditional multi-pass compiler architecture with a number of standard optimizations. The overall architecture provided by `syntax-spec` ensures that code generated by extensions benefits from optimizations, too. Most notably, our compiler works carefully around host-language code contained in miniKanren goals to optimize where possible while accounting for the host language code's unknown behavior. In some sense, the details of the optimizations and the performance they yield are not very exciting, but they demonstrate that a `syntax-spec` DSL can be equipped with a standard compiler back-end. To underline how effective the compiler is, we show at the end of this section that our compiler produces substantial and sometimes asymptotic performance improvements.

5.1 Optimizations for miniKanren

Following the nanopass approach [30, 45], our compiler back-end consists of many small passes. We group them into four major steps to discuss their effects.

Constant Folding. The first major pass implements constant folding. It tracks statically-known equational information with a compile-time substitution data structure. The compiler uses information gained from earlier conjuncts to simplify subsequent ones. When unifications are statically

guaranteed to succeed trivially or fail, the compiler simplifies them to just `succeed` or `fail` respectively. This pass also decomposes complex equations into conjunctions of simple "variable on the left" ones.

Dead Code Elimination. Dead code elimination requires several small passes. The first one removes any code dominated in control flow by a `fail` (usually introduced by constant folding). The second pass finds equations that are statically known to succeed and whose execution does not further constrain the domains of external variables, and it replaces all such equations with `succeed`. The remaining passes simplify conjunctions with trivial `succeed`s and remove from `fresh`es the bindings of unused logic variables.

Unification Analysis. Two further passes annotate unifications with information that allows the code generator to produce more specialized code [55]. The first of these two employs abstract interpretation to mark unifications for which it is safe to skip an occurs check [49]. An occurs check is generally required to forbid cyclic terms and ensure soundness of deductions [35], but it is expensive: the cost is linear in the size of the run-time terms being unified. Unification in miniKanren always includes the occurs check, but it is unnecessary when the compiler can statically determine that the equation does not introduce a cycle. The analysis correspondingly uses an abstract domain that records whether each variable is fresh, is known to refer to a limited set of other variables, or has a wholly unknown value. The second unification analysis pass marks the first reference to a newly introduced logic variable. This reference can be compiled more efficiently because the specializer knows the variable is fresh.

Specialization. Normally, miniKanren performs unification via a runtime operation that recursively inspects the structure of two terms. Our code generator specializes unification to any syntactically evident structure. For example, in

```
(== x (cons first (cons second rest)))
```

the term on the right-hand-side always has at least two pairs. The unification procedure's dispatch can thus be unfolded and simplified for this portion of the match. The code generator also employs the annotations from unification analysis to generate calls to a version of unification without the occurs check when possible.

5.2 Extensions Get Optimized Too

DSL programmers who extend the language syntactically also benefit from this compiler pipeline, following Dybvig's "macro writer's bill of rights" [18]. Dybvig proposes that compilers should guarantee to perform certain optimizations such as constant folding and dead-code elimination, allowing macro authors to write simple transformations that may sometimes introduce unnecessary indirections or add dead code—without sacrificing performance. Through `syntax-spec`, our DSL's architecture allows us to offer DSL programmers these rights.

To show how extensions automatically benefit from our compiler pipeline, we step through an example. Below is `leo`, an inequality relation on Peano numerals adapted from Rozplokhas and Boulytchev [44]:

```
(defrel/matche (leo x y)
  [('Z y)]
  [((cons 'S x1) (cons 'S y1)) (leo x1 y1)])
```

Our `leo` definition uses `defrel/matche`, which in turn uses `matche`. The simplistic implementation of `matche` from Section 4.1 introduces inefficiencies. Figure 15 shows `leo` program after each of three major steps, using plain source code for readability.

```
(defrel (leo x y)
  (fresh (ls)
    (conj
      (== ls (cons x (cons y '())))
      (disj
        (fresh (y^)
          (== (cons 'Z (cons y^ '()))
              ls))
        (fresh (x1 y1)
          (conj
            (==
              (cons (cons 'S x1)
                    (cons (cons 'S y1)
                          '()))
              ls)
            (leo x1 y1)))))))
```

```
(defrel (leo x y)
  (fresh (ls)
    (conj
      (== ls (cons x (cons y '())))
      (disj
        (fresh (y^)
          (conj (== x 'Z) succeed))
        (fresh (x1 y1)
          (conj
            (conj
              (== x (cons 'S x1))
              (conj
                (== y (cons 'S y1))
                succeed))
            (leo x1 y1)))))))
```

```
(defrel (leo x y)
  (fresh ()
    (disj
      (fresh ()
        (== x 'Z))
      (fresh (x1 y1)
        (conj
          (conj
            (== x (cons 'S x1))
            (== y (cons 'S y1)))
          (leo x1 y1))))))
```

Fig. 15. The `leo` program after each of: expansion (left); constant folding (middle); and dead-code elimination (right). Underlines indicate the parts of the program that are changed in the next frame. In the first frame, they highlight the unifications that are simplified by constant folding. In the second, they show which portions of the program are removed by dead code elimination.

The macro expander desugars the Racket `defrel/matche` macro into a `defrel` with a `matche` in the body. The expander further desugars `matche` and all the miniKanren surface syntax forms together into our core language. These expansions result in a program with a number of unnecessary indirections, shown in the left-hand frame of Figure 15. The expansion introduces the intermediate variable `ls`, a list of all the relation's arguments. Each disjunct unifies `ls` with the term compiled from the pattern. Through this indirection, the generated code naively unifies lists of all arguments against entire patterns; this is wasteful when the terms are statically known to share structure.

Constant folding and dead code elimination address these inefficiencies. The middle frame of Figure 15 shows the result of constant folding. The pass eliminates two references to `ls` and also simplifies the remaining equations. Dead code elimination cleans up after the constant folding pass by removing trivial pieces. By the end of dead code elimination, shown in the right-hand frame, the compiler optimizes away all uses of the unnecessary variable `ls`. It retains `(fresh () ...)` nodes in the final frame of Figure 15 even after dead code elimination because the `faster-minikanren` run-time system establishes interleaving points for its search at `fresh`. We therefore keep these nodes to achieve answer order equivalence with the existing implementation.

Keep et al., who first introduced `matche`, write that one of their primary aims is "to generate code that will perform at least as well as if the generated code had been written by a human" [29]. Our simplistic implementation of `matche` from Section 4.1 initially seems to fall short of that aim, but macro-expanding to the input language of an optimizing compiler solves the problem with no extra effort on the behalf of the macro author. Our compiler removes `matche`'s unnecessary indirections whether the `matche` comes from the source program or through the expansion of another macro like `defrel/matche`. Without these optimization guarantees, the DSL extension programmer would have to inspect the entire macro stack to ensure that a new macro will generate performant code. An optimizing compiler for the core language relieves the extension programmer of this burden. In short, the benefits stack up as the layers of languages and extensions do.

5.3 Optimizing at the Boundary with Racket

Compiler correctness and performance also entail preserving certain properties of mixed miniKanren and Racket language programs. As mentioned in Section 4.2, we can understand the DSL-host language combination as a Matthews-Findler multi-language. When adding extensibility and a

host-language interface, the key is to hit a "sweet spot" of adding the desired expressive power [20] without losing the ability to reason about the DSL program as something more than mere host language code.

Our multi-language hits such a sweet spot. It increases the expressive power of extensions without exposing internal implementation details that would prevent semantics-preserving optimization. Consider programs of the following shape:

```
(conj
  (fresh (x y)
    (== x y))
  (goal-from-expression
    #| ... unknown racket code ... |#))
```

Our optimizer's constant propagation and dead code elimination transform the first conjunct into:

```
  (fresh () succeed)
```

After all, no matter what Racket code is in the following goal, it cannot observe the fact that the optimizer has removed the allocation of those logic variables. Alternative host-interface designs could render such optimizations impossible. For example, if the Racket code were able to access the data structure storing the current values of all logic variables and enumerate them all, its behavior could change when our compiler removes otherwise-dead variables.

At the same time, the explicit boundaries between the two languages enable the optimization passes to rein in their transformations to account for the unknown behavior of the Racket code. For example, the occurs check elimination pass is partially limited in this program:

```
(fresh (x y a b)
  (== x (cons '5 '6))
  (goal-from-expression
    #| unknown racket code |#)
  (== a b)
  (== y x))
```

The miniKanren compiler needs to run before Racket code in boundary forms is expanded and available for analysis. Therefore, the compiler must treat the body of each such form as having potentially arbitrary behavior. The unknown Racket could access and, via `expression-from-goal`, further constrain any of the variables in scope. For example, it could assign the value `(cons '1 a)` to the variable `b`. Thus the unification `(== a b)` still needs an occurs check to prevent a cyclic term. However, the optimizer does not lose all information following a `goal-from-expression`. Code via the host interface can only modify the state by executing goals constructed via `expression-from-goal` and only in ways that accord with the usual miniKanren semantics. In particular the optimizer can be sure that the state is only extended in a monotonic way, that the assignment to the variable `x` is unchanged, and hence an occurs check for the final unification is unnecessary.

The `syntax-spec` framework facilitates hitting the aforementioned expressive power sweet spot by structuring the DSL definition as a multi-language where the DSL and host interact only at the specified boundary. The DSL has a separate grammar only connecting to the host at specified `host-interface` and `racket-expr` positions. Name bindings work similarly. DSL names like `term-variables` belong to separate binding classes. They may be used in Racket code only if a value translation is defined using `with-reference-compilers`. These choices ensure we can have the interactions that we want and prevent unexpected interaction. The multi-language structure could also provide the basis for formal reasoning about the correctness of our miniKanren compiler, along the lines of Perconti and Ahmed [40].

Table 1. Our compiler's performance results on a selection of miniKanren tests. The faster-mK benchmark column reports time in milliseconds; subsequent columns report speedup ratios over that column. Larger numbers report better speedups. Tests were run on a M1 Max Macbook Pro with 64GB RAM.

Benchmark	faster-mK	no opts	prop only	dead code	occurs check	specialization and overall
Occurs check						
`leo` 8000	209	1	1	1	52.25	69.67
appendo w/2 lists	327	0.99	1	1.01	81.75	109
Relational arithmetic						
`logo`	437	0.99	1	0.98	1.08	1.44
`four fours` of 256	77	0.95	0.97	0.94	1.01	1.26
`fact x` = 720	122	1.04	1.03	1.01	1.18	1.56
Relational interpreters						
one quine	962	0.99	0.96	0.97	1.01	1.01
9,900 `(I love you)`s	1378	0.96	0.95	0.98	1.01	1.04
append synthesis	252	1	1	0.98	1.33	1.81
dynamic and lexical	18	1	1.06	1	1.2	1.5
four thrines	683	0.98	1	1	1.03	1.06
countdown from 2 in λ-calc	64	0.97	1	1	6.4	7.11

5.4 Benchmarks and Results

To evaluate our optimizations' effectiveness, we assembled a benchmark suite and measured the speedup produced by each important category of compiler optimization. The point though is not to prove the effectiveness of our particular optimizations, but to show that the architecture enabled by `syntax-spec` accommodates an aggressive compiler.

5.4.1 Benchmark Suite. We assembled a benchmark suite from examples in several papers on pure relational programming in miniKanren.

Occurs Check. Rozplokhas and Boulytchev [44] analyzed the asymptotic complexity of a variety of miniKanren programs. They note that the occurs check sometimes contributes substantially to the asymptotic cost. Because we expect that our optimizer can remove some of these checks, we adopt two of these programs as benchmarks. One uses the `leo` program introduced in Section 5.1 and searches for a Peano numeral greater than 8000. The second program uses a two-place append relation that connects a pair of lists with the result of appending the first to the second. This test runs the relation with a pair of two large ground lists.

Relational Arithmetic. The second set of benchmark programs exercise the miniKanren relational arithmetic suite introduced by Kiselyov et al. [31]. The first of these solves a difficult logarithm. The second program solves the `four-fours` puzzle [3] for 256, and the third searches for a number with a factorial of 720.

Relational Interpreters. The third and largest suite of programs use relational interpreters to synthesize programs with specified behaviors. The first example replicates the inaugural application of relational interpreters from Byrd et al. [10]: generating a Scheme quine. The next several are from Byrd et al. [9]. They include deriving 9900 expressions that evaluate to `(I love you)`, example-based synthesis of part of the standard append function, and synthesizing programs that evaluate differently under different semantics. One more program from Byrd et al. [9] computes a thrine, a 3-cycle of different programs that evaluate to one another. The last program in this suite uses a relational interpreter to evaluate a large known program.

5.4.2 Results. Table 1 shows the performance improvements on our benchmark suite. To bring across what benefits each optimization realizes, we show the time to compute each program with the baseline `faster-miniKanren` implementation, and the speedup from our compiler with different configurations of optimization passes enabled. The header of each column indicates the additional optimization added in that configuration; each configuration includes the optimizations enabled in the columns to its left. Thus, the final "specialization" column also reports the overall speedup achieved by the compiler.

The occur-check removal is far and away the most helpful optimization. In addition to the examples where we expected to see improvement, we found it was also important for the relational interpreter benchmark with a large ground program ("countdown from 2 in λ-calc"). Unification specialization is usually helpful or harmless, but occasionally produces slowdowns by increasing the size of generated code. Constant propagation and dead-code elimination do not confer much benefit. Their performance is artificially limited by our desire to maintain answer order equivalence with `faster-miniKanren`, which prevents us from removing dead code that impacts search order. We validate the correctness of our compiler against a larger test suite, and on these tests we do achieve answer-order equivalence with `faster-miniKanren`.

6 DSLs As Host Language Extensions, Not Embeddings

This effort extends the investigation of macro-extensible DSLs, an approach with a long tradition in the Scheme and Racket communities [48, 51]. Work by Ballantyne et al. [5, 6] puts the approach on a systematic foundation, and their `syntax-spec` DSL makes constructing such DSLs easy. This pearl extends that line of thinking with an investigation of the synthesis of DSL compiler optimizations, multi-language boundaries with the host language, and DSL extensibility. Optimizations improve extensibility by freeing extension authors from worrying about low-level details of the code they generate. A rich interface with the host language improves extensibility by increasing the expressive power of the DSL. And combining these features requires ensuring that optimizations account for the interaction of host and DSL code in a multi-language setting. Our chosen example, miniKanren, epitomizes a class of DSLs (e.g. PEG [6], Qi [28]) that benefit from this confluence of techniques.

The task of DSL embedding suggests re-using some facets of the host in the DSL that is being built, and there is a widely explored space of trade-offs [37] surrounding such linguistic re-use. This design space is anchored at two ends by the *deep* and *shallow* embeddings. Both re-use host-language syntax, but they differ to what degree they re-use host language semantics. Shallow embeddings make it trivial for the implementer, or an ordinary DSL user, to extend the DSL and to integrate host-language code, because DSL code is itself simply host language code. In shallow embeddings of miniKanren, each syntactic form is realized in the host language as either a call to a function or via an individual host-language macro that expands to runtime functions and to host-language binding forms such as `lambda`. The most popular implementations of miniKanren are shallow embeddings [4, 25, 33], and take advantage of host-language code and extensibility. However, there is no DSL compiler that gets an overall view of the program, so optimizing compilation is impossible.

By contrast, deep embeddings can offer whole-program compilation, and miniKanren researchers have developed optimizations using a variety of deep embeddings [34, 56, 57]. In deep embeddings, host language code constructs a datatype representing miniKanren abstract syntax. A DSL compiler executing at the host language's runtime takes the abstract syntax as input and produces host language code as output. No existing implementation of miniKanren, however, reconciles the benefits of both implementation techniques.

Let us turn to the question of how the approach we describe relates to other attempts at reconciling extensibility and optimizing DSL compilers. One fruitful approach is to layer a shallow embedding on top of a deep embedding [19, 26, 50]. The shallow embedding provides integration with the host language syntax and static semantics. It also achieves extensibility: extensions are host language function definitions, whose applications run in the host language. The semantic domain of the shallow embedding is the datatype of the deep embedding. A DSL optimizing compiler processes the data representation of the deep embedding. Our approach has parallels to this layering. DSL extensions via macros are analogous to functions defining extensions in a shallow embedding, but the process that generates DSL core language code is macro expansion rather than host-language evaluation.

Another approach relies on reflection to extract code from a shallow embedding for compilation [1, 38, 43, 46, 47, 58]. DSL programs are written as host language code calling functions that represent DSL forms, just as in a shallow embedding. These functions may be defined as normally in a shallow embedding, or they may instead be mere stubs. This design works because the host language code is not actually executed in the normal way. Instead, a quotation or compile-time reflection mechanism is used to extract a representation of the host language code. A DSL compiler provides an alternate interpretation of the code, recognizing applications of the stub functions as DSL constructs to compile. This DSL compiler thus has control over the complete DSL program, so it may perform analyses and optimizations. Extensibility may be achieved by employing a normalizer for the host language, either built-in to the reflection mechanism or separately defined. The normalizer is configured to avoid reducing calls to the stub functions representing the DSL core language, but it reduces other calls. Just as function calls representing uses of extensions evaluate in the host language in a shallow embedding, they reduce during normalization and leave behind only calls that belong to the DSL core language.

Both of these approaches rely on encoding the DSL in the host language syntax and static semantics. Indeed, re-use of these components is a key, separate motive for embedding-based implementation techniques, beyond extensibility. This is particularly true when the motive is to reuse the host language type system, as in the case of re-using Haskell's type system to create a typed logic programming language [14, 27], or embedding a DSL in a theorem prover in order to support verification [58]. However, reuse of the host syntax and static semantics can come at a serious cost when the DSL has significant differences with the host. If the DSL's type system cannot be expressed in the host type system, there is no benefit to re-use. If a DSL's re-interpretation of the host language only supports a small subset of its syntax, DSL users may have trouble understanding what constructs they can use in DSL code. Finally, DSL compiler authors must contend with the complexity of the host language and its representation in the reflection system, including portions irrelevant to their DSL.

Our approach to DSLs and extension makes particular sense in cases where the DSL and host differ in important ways. DSLs may come with their own new syntax, including binding structure, and new static semantics. Macros, too, define new syntax and binding structure rather than reuse that of the host. This approach avoids any impedance when either using or implementing the DSL.

7 Bringing Macro-extensible Hosted DSLs to a Language Near You

Bringing our approach to extensibility to other hosts means addressing three further challenges: non-S-expression syntax, static checking, and IDE services.

Related efforts in these directions show early promise. A variety of designs [2, 17, 23, 41, 42, 53] have explored the integration of macros with Algol-style language syntax. A `syntax-spec`-like metalanguage in a host with such traditional syntax would need to integrate these ideas into its means for specifying grammar. In a typed host language, creating a DSL means defining a

typed multi-language. A type-system specification language like Turnstile [11, 12] or Statix [54] could serve for specifying the type system for the DSL facet of that multi-language. Reflection mechanisms as found in Scala 2 macros [8], elaborator reflection [13], and Klister's [7] stuck macros would connect the host and DSL type systems. Specifically, the DSL type system could use these reflection mechanisms to access the host typing environment and the type expected by the host context at boundaries. Separate but interacting DSL and host-language type checkers would enable DSL type systems that are not expressible via embedding in the host type system; e.g. a DSL with affine types in a conventional host [52]. Contracts at the multi-language boundary would remain useful to enforce invariants of the DSL that cannot be statically guaranteed by host-language types.

A language is more than syntax and semantics; to a modern programmer, it includes an IDE or language server. The implementation of `syntax-spec` cooperates with Racket's macro system to convey information about binding structure to Racket IDEs such as DrRacket. In a typed host, such information should extend to the types associated with bindings derived from the DSL's type system specification. Racket IDEs can rely on their understanding of the S-expression syntax shared by Racket and all `syntax-spec` DSLs to provide other services. In a language where a DSL specification is allowed more flexibility to define new lexical syntax, the IDE would need information from that grammar to provide syntactic services such as highlighting and balanced brace matching. Thus the channel of communication between a DSL definition metalanguage and an IDE in such languages needs to be wider, possibly using mechanisms such as those provided in the Lean 4 macro system [53].

Demonstrating the combination of all these ideas in a `syntax-spec`-like metalanguage remains as future work. However, these lines of inquiry point towards bringing the synthesis of easy extension, interoperation with the host language, and optimizing compilation to a wider variety of languages.

Acknowledgments

We thank Matthias Felleisen, William E. Byrd, Nada Amin, Greg Rosenblatt and the anonymous reviewers for their feedback and suggestions. The research of Michael Ballantyne and Mitch Gamburg has been partially supported by two NSF grants (CCF-2007686, CCF-2116372). The research of Jason Hemann has been partially supported by NSF grant CCF-2348408.

References

[1] Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding domain-specific languages. In *Proc. Symposium on Haskell (Haskell '09)*. 37–48. https://doi.org/10.1145/1596638.1596644

[2] Jonathan Bachrach and Keith Playford. 1999. D-Expressions: Lisp Power, Dylan Style. https://people.csail.mit.edu/jrb/Projects/dexprs.pdf

[3] W. W. Rouse Ball. 1914. *Mathematical Recreations and Essays (6th Edition)*. MacMillan & Co., Limited.

[4] Michael Ballantyne. 2024. faster-minikanren. https://github.com/michaelballantyne/faster-minikanren.

[5] Michael Ballantyne and Matthias Felleisen. 2023. Injecting Language Workbench Technology into Mainstream Languages. In *Eelco Visser Commemorative Symposium*. 3:1–3:11. https://doi.org/10.4230/OASIcs.EVCS.2023.3

[6] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for Domain-Specific Languages. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 229 (11 2020). https://doi.org/10.1145/3428297

[7] Langston Barrett, David Thrane Christiansen, and Samuel Gélineau. 2020. Predictable Macros for Hindley-Milner. In *The Workshop on Type-Driven Development (TyDe '20)*. https://davidchristiansen.dk/pubs/tyde2020-predictable-macros-abstract.pdf

[8] Eugene Burmako. 2013. Scala macros: let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proc. Workshop on Scala (Scala '13)*. Article 3. https://doi.org/10.1145/2489837.2489840

[9] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (8 2017). https://doi.org/10.1145/3110252

[10] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proc. Workshop on Scheme and Functional Programming (Scheme '12)*. 8–29. https://doi.org/10.1145/2661103.2661105

[11] Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2019. Dependent type systems as macros. *Proc. ACM Program. Lang.* 4, POPL (Dec. 2019). https://doi.org/10.1145/3371071

[12] Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type systems as macros. In *Proc. Principles of Programming Languages (POPL '17)*. 694––705. https://doi.org/10.1145/3009837.3009886

[13] David Christiansen and Edwin Brady. 2016. Elaborator reflection: Extending Idris in Idris. In *Proc. International Conference on Functional Programming (ICFP '16)*. 284––297. https://doi.org/10.1145/2951913.2951932

[14] Koen Claessen and Peter Ljunglöf. 2001. Typed Logical Variables in Haskell. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 37. https://doi.org/10.1016/S1571-0661(05)80544-4

[15] Ryan Culpepper. 2012. Fortifying macros. *Journal of Functional Programming* 22, 4-5 (8 2012), 439–476. https://doi.org/10.1017/s0956796812000275

[16] Ryan Culpepper and Matthias Felleisen. 2010. Fortifying macros. In *Proc. International Conference on Functional Programming (ICFP '10)*. 235––246. https://doi.org/10.1145/1863543.1863577

[17] Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. 2014. Sweeten your JavaScript: Hygienic macros for ES5. In *Proc. Symposium on Dynamic Languages (DLS '14)*. 35–44. https://doi.org/10.1145/2661088.2661097

[18] R. Kent Dybvig. 2004. The guaranteed optimization clause of the macro-writer's bill of rights. https://www.youtube.com/watch?v=LIEX3tUliHw Presented at Dan Friedman's 60th birthday conference.

[19] Conal Elliott, Sigbjørn Finne, and Oege de Moor. 2003. Compiling embedded languages. *Journal of Functional Programming* 13, 3 (2003), 455–481. https://doi.org/10.1017/S0956796802004574

[20] Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17, 1-3 (1991), 35–75. https://doi.org/10.1016/0167-6423(91)90036-W

[21] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proc. International Conference on Functional Programming (ICFP '02)*. 48––59. https://doi.org/10.1145/581478.581484

[22] Matthew Flatt. 2002. Composable and compilable macros: You want it when?. In *Proc. International Conference on Functional Programming (ICFP '02)*. 72––83. https://doi.org/10.1145/581478.581486

[23] Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. 2023. Rhombus: A New Spin on Macros without All the Parentheses. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 242 (Oct. 2023). https://doi.org/10.1145/3622818

[24] Aleksandra Foksinska, Camerron M Crowder, Andrew B Crouse, Jeff Henrikson, William E Byrd, Gregory Rosenblatt, Michael J Patton, Kaiwen He, Thi K Tran-Nguyen, Marissa Zheng, et al. 2022. The precision medicine process for treating rare disease using the artificial intelligence tool mediKanren. *Frontiers in Artificial Intelligence* 5 (2022). https://doi.org/10.3389/frai.2022.910216

[25] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition*. The MIT Press.

[26] Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: Deep and shallow embeddings (functional Pearl). In *Proc. International Conference on Functional Programming (ICFP '14)*. 339–347. https://doi.org/10.1145/2628136.2628138

[27] Ralf Hinze. 1998. Prological Features in a Functional Setting Axioms and Implementation.. In *Fuji International Symposium on Functional and Logic Programming (FLOPS '98)*. 98–122. https://www.cs.ox.ac.uk/ralf.hinze/publications/FLOPS98.ps.gz

[28] Siddhartha Kasivajhula and `drym.org`. 2024. Qi: An Embeddable Flow-Oriented Language. https://github.com/drym-org/qi.

[29] Andrew W Keep, Michael D Adams, Lindsey Kuper, William E Byrd, and Daniel P Friedman. 2009. A pattern matcher for miniKanren or how to get into trouble with CPS macros. In *Proc. Workshop on Scheme and Functional Programming (Scheme '09)*. 37–45. https://digitalcommons.calpoly.edu/csse_fac/83

[30] Andrew W. Keep and R. Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *Proc. International Conference on Functional Programming (ICFP '13)*. 343––350. https://doi.org/10.1145/2500365.2500618

[31] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *Proc. International Symposium on Functional and Logic Programming*. 64–80. https://doi.org/10.1007/978-3-540-78969-7_7

[32] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proc. International Conference on Functional Programming (ICFP '05)*. 192–203. https://doi.org/10.1145/1086365.1086390

[33] Dmitrii Kosarev and Dmitry Boulytchev. 2018. Typed Embedding of a Relational Language in OCaml. In *Proc. ML Family Workshop (ML/OCAML 2016)*. https://doi.org/10.4204/EPTCS.285.1

[34] Peter Lozov and Dmitry Boulytchev. 2021. Efficient fair conjunction for structurally-recursive relations. In *Proc. Partial Evaluation and Program Manipulation (PEPM 2021)*. https://doi.org/10.1145/3441296.3441397

[35] Kim Marriott and Harald Søndergaard. 1989. On Prolog and the occur check problem. *ACM SIGPLAN Notices* 24, 5 (1989), 76–82. https://doi.org/10.1145/66068.66075

[36] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *Proc. Principles of Programming Languages (POPL '07)*. 3––10. https://doi.org/10.1145/1190216.1190220

[37] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (dec 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[38] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: Quoted domain-specific languages. In *Proc. Partial Evaluation and Program Manipulation (PEPM '16)*. 25–36. https://doi.org/10.1145/2847538.2847541

[39] David Nolen, Rich Hickey, and contributors. 2020. *clojure/core.logic: A logic programming library for Clojure and ClojureScript*. https://github.com/clojure/core.logic

[40] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *European Symposium on Programming Languages and Systems (ESOP '14)*. 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

[41] The Rust Project. 2024. The Rust Reference: Procedural Macros. https://doc.rust-lang.org/reference/procedural-macros.html

[42] Jon Rafkind and Matthew Flatt. 2012. Honu: Syntactic extension for algebraic notation through enforestation. In *Proc. Generative Programming and Component Engineering (GPCE '12)*. 122–131. https://doi.org/10.1145/2371401.2371420

[43] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: Linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* 25, 1 (01 Mar 2012), 165–207. https://doi.org/10.1007/s10990-013-9096-9

[44] Dmitry Rozplokhas and Dmitry Boulytchev. 2021. A Complexity Study for Interleaving Search. In *Proc. miniKanren and Relational Programming Workshop (miniKanren '21)*. http://minikanren.org/workshop/2021/minikanren-2021-final7.pdf

[45] DiPanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2005. Educational Pearl: A Nanopass framework for compiler education. *Journal of Functional Programming* 15, 5 (2005), 653––667. https://doi.org/10.1017/S0956796805005605

[46] Maximilian Scherr and Shigeru Chiba. 2014. Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding. In *Proc. European Conference on Object-Oriented Programming (ECOOP '14)*. 385–410. https://doi.org/10.1007/978-3-662-44202-9_16

[47] Amir Shaikhha, Vojin Jovanovic, and Christoph Koch. 2018. A Compiler-Compiler for DSL Embedding. (Aug. 2018). arXiv:1808.01344

[48] Olin Shivers. 2005. The anatomy of a loop: A story of scope and control. In *Proc. International Conference on Functional Programming (ICFP '05)*. 2–14. https://doi.org/10.1145/1086365.1086368

[49] Harald Søndergaard. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In *Proc. European Symposium on Programming (ESOP '86)*. 327–338. https://doi.org/10.1007/3-540-16442-1_25

[50] Josef Svenningsson and Emil Axelsson. 2013. Combining Deep and Shallow Embedding for EDSL. In *Proc. Trends in Functional Programming (TFP '12)*. 21–36. https://doi.org/10.1007/978-3-642-40447-4_2

[51] Sam Tobin-Hochstadt. 2011. Extensible Pattern Matching in an Extensible Language. (2011). arXiv:1106.2578

[52] Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *Proc. European Symposium on Programming (ESOP '10)*. 550–569. https://doi.org/10.1007/978-3-642-11957-6_29

[53] Sebastian Ullrich and Leonardo de Moura. 2022. Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages. *Logical Methods in Computer Science* 18, 2 (April 2022). https://doi.org/10.46298/lmcs-18(2:1)2022

[54] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). https://doi.org/10.1145/3276484

[55] Peter Van Roy. 1994. 1983–1993: The wonder years of sequential Prolog implementation. *The Journal of Logic Programming* 19-20 (1994), 385–441. https://doi.org/10.1016/0743-1066(94)90031-0

[56] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2020. An Empirical Study of Partial Deduction for miniKanren. In *Proc. miniKanren and Relational Programming Workshop (miniKanren '20)*. http://minikanren.org/workshop/2020/minikanren-2020-paper2.pdf

[57] Ekaterina Verbitskaia, Igor Engel, and Daniil Berezun. 2023. Semi-Automated Direction-Driven Functional Conversion. In *Proc. miniKanren and Relational Programming Workshop (miniKanren '23)*. http://minikanren.org/workshop/2023/minikanren23-final2.pdf

[58] Artjoms Šinkarovs and Jesper Cockx. 2021. Choosing is Losing: How to combine the benefits of shallow and deep embeddings through reflection. arXiv:2105.10819

Received 2024-02-28; accepted 2024-06-18