Fast Sequences in Racket

ANONYMOUS AUTHOR(S)

Racket provides `for` loops with macro-extensible sequence expressions. Sequence macros offer better performance than dynamic sequence implementations, but they are complicated to define and Racket offers no support for creating new sequence macros by combining existing ones. In this paper, we develop such support in the form of sequence combinator macros and a general comprehension form. These utilities are implemented by manipulating compile-time records that contain binders and expressions; the binding relationships between the components are not trivial. We present an informal notation for macros, bindings, and types, which we used to diagnose and solve scoping problems in our implementation.

1 SEQUENCES AND FOR-LOOPS IN RACKET

Racket provides a family of `for` loops based on the eager comprehensions SRFI for Scheme (Egner 2005). A loop consists of a `for`-variant, clauses that describe iteration, and a body. For example, the following expression iterates over the elements of the list, squares them, and returns their sum:

```
> (for/sum ([x (list 1 2 3 4 5)])
    (sqr x))
55
```

The `for/list` variant would collect the squared values into a list; `for/product` would compute their product; the basic `for` variant would discard them; and so on.

Iteration is controlled by *sequences*. Values such as lists, vectors, and hashes implement the sequence generic interface, so they can be used in `for`-loop clauses. The generic interface is dispatched at run time, however, so it is slower than a hand-written recursive function using datatype-specific operations for extracting data. For better performance, Racket also offers sequence macros, such as `in-list` and `in-vector`. When a sequence macro is used directly in the right-hand side of a `for` clause, the compiler generates specialized code for that kind of sequence, which often enables additional optimizations such as inlining. Here are some examples:

```
> (for/sum ([x (in-list (list 1 2 3 4 5))])
    x)
15
> (for ([(key val) (in-hash (hash 'a 1 'b 2 'c 3))])
    (printf "~s = ~s; " key val))
c = 3; b = 2; a = 1;
```

In the second example, each *element* of the sequence contains two values, and each value is bound to the corresponding *iteration variable* (`key` and `val`).[1]

We call an application of a sequence macro a *fast sequence expression* or simply a *fast sequence* to distinguish it from an expression that produces a *dynamic sequence* value. A fast sequence is only treated specially if it is used in a *fast sequence context*, such as the right-hand side of a `for` clause; otherwise, it acts as an ordinary expression and produces a dynamic sequence.

Racket provides an assortment of combinators—such as mapping and filtering operations—for working with dynamic sequences, but Racket does not provide analogous combinators for fast sequences. In other words, dynamic sequences are expressive but less efficient, whereas fast sequences provide high performance but they lack expressiveness. This gap motivates our work.

[1]The iteration order of hashes is not specified.

Author's address: Anonymous Author(s).

To illustrate the dichotomy between expressiveness and efficiency, consider the task of calculating a sum and a product of squares of prime numbers up to some number, like 10000. The simplest solution is to use lists. In Racket, we can use `range`, `filter`, and `map` to define a function that produces a list of the desired numbers:

```
> (define (squares-of-primes-up-to n)
    (map sqr
     (filter prime?
      (range n))))
```

Then we use `squares-of-primes-up-to` to solve both problems:

```
> (for/sum ([(x) (squares-of-primes-up-to 10000)]) x)
37546387960
> (void
   ; a very long number
   (for/product ([(x) (squares-of-primes-up-to 10000)]) x))
```

This solution has a downside: it eagerly allocates a big intermediate list. That can be avoided by using the sequence macro `in-range` along with `sequence-filter` and `sequence-map`, which generalize the list-specific `filter` and `map` functions to sequences:

```
> (define (squares-of-primes-up-to n)
    (sequence-map sqr
     (sequence-filter prime?
      (in-range n))))
```

The rest of the solution remains the same. The sequence produced by `in-range` is lazy, so it eliminates the second downside of the previous solution. Unfortunately, even though `in-range` is a sequence macro, it is not used in a fast sequence context here, because `sequence-filter` is an ordinary function, so it fails to improve performance.

If we abandoned the idea of defining an intermediate abstraction, we could simply write two `for` loops, using `prime?` in a `#:when` clause to filter iterations and calling `sqr` in the body:

```
> (for/sum ([x (in-range 10000)]
            #:when (prime? x))
    (sqr x))
37546387960
; likewise for for/product
```

This code is efficient, but it sacrifices the reusable abstraction.

In fact, Racket does provide an interface for defining new sequence macros, but it is cumbersome because it lacks composition facilities. This naturally motivates new tools for making it easier to define new fast sequence forms via composition. We begin by defining `fast-sequence-map` and `fast-sequence-filter`; unlike `sequence-map` and `sequence-filter`, their second argument is a fast sequence context. We use them as follows:

```
> (define-sequence-rule (squares-of-primes-up-to n)
    (fast-sequence-map sqr
     (fast-sequence-filter prime?
      (in-range n))))
```

The body of an ordinary Racket function is not a fast sequence context, so we must define our abstraction as a sequence macro. No changes are required to the clients of `squares-of-primes-up-to` (the `for/sum` and `for/product` expressions); they automatically gain the benefit of the chain of fast sequence expressions. That is, `define-sequence-rule`, `fast-sequence-map`, and `fast-sequence-filter` have allowed us to easily define a reusable sequence abstraction by building upon an existing fast sequence form (`in-range`).

This paper discusses the design and implementation of our support for fast sequences—an example of macros for a macro-extensible DSL (Ballantyne et al. 2020; Flatt et al. 2012) in Racket. Section 2 presents our first abstraction facilities based on `map` and `filter`; Section 3 explains their implementation in Racket. The syntactic forms of the DSL and their compile-time intermediate representations have nontrivial scoping rules, so we introduce an informal notation and discipline for keeping track of scopes. Section 4 generalizes our previous solutions and adds a new capability: nesting. The scoping rules for nesting present an additional challenge, and we show how to use Racket's compile-time API to implement it correctly.

2 MAPPING AND FILTERING

This section explains how we implement `fast-sequence-map` and `fast-sequence-filter` in terms of Racket's interface for sequence macros. We explain that interface by showing how `for` loops are compiled to recursive functions.

2.1 fast-sequence-map

A `for` loop in Racket has *clauses* and a *body*. A clause contains a list of *iteration variables* and a sequence expression. If the clause contains a single iteration variable, it may be written without parentheses, but Racket converts it to a singleton list before processing the clause. For example:

```
> (for ([x (in-list '(1 2 3))])
    (printf "~a " x))
1 2 3
```

This example has a single clause, `[(x) (in-list '(1 2 3))]`, which binds the iteration variable `x` in the body `(printf "~a " x)`.

In Racket, `for` is not a primitive syntactic form; rather, it is a macro that expands into a call of a recursive function. That function is formed by three parts. Firstly, its shape is dictated by the variant of the `for` expression (e.g., `for`, `for/sum`). Secondly, it uses the information that comes from the `for` loop clauses: namely, about how to start looping, how to decide whether to stop looping, what values to bind to the iteration variables and auxiliary variables, and how to recur. Finally, the third part is the `for` loop body. We show a simplified version of the recursive function for the example above:

```
> (let loop ([lst '(1 2 3)]) ; how to start iteration
    (when (pair? lst) ; how to decide whether to stop
      (let-values ([(x) (car lst)] ; what to bind (iteration variable)
                   [(rest) (cdr lst)]) ; what to bind (auxiliary variable)
        (printf "~a " x) ; the loop body
        ; how to recur
        (loop rest))))
1 2 3
```

The first part is a *loop skeleton*, which comes from the `for` loop variant and has blanks. For the current example, it tells the following structure, where the blanks are highlighted:

```
(let loop ([loop-id loop-expr] ...)
  (when pos-guard
    (let-values ([(inner-id ...) inner-expr] ...)
      loop-body
      (loop loop-arg ...))))
```

This is the loop skeleton of `for`. The loop skeletons of other `for` variants are slightly different.

The second part is a compile-time record for each `for` loop clause, the clause's intermediate representation. Each record is produced by the sequence macro in the clause's right-hand side. For example, the clause `[(x) (in-list '(1 2 3))]` is translated by `in-list` into the following:

$$\begin{aligned} \textit{loop-bindings} &= \langle\ \texttt{lst} \leftarrow \texttt{'(1 2 3)}\ \rangle \\ \textit{pos-guard} &= \texttt{(pair? lst)} \\ \textit{inner-bindings} &= \langle\ \texttt{(x)} \leftarrow \texttt{(car lst)}, \\ &\qquad\quad \texttt{(rest)} \leftarrow \texttt{(cdr lst)}\ \rangle \\ \textit{loop-args} &= \langle\ \texttt{rest}\ \rangle \end{aligned}$$

The components of this record are the ingredients that fill in the blanks of the loop skeleton. This record's bindings include the iteration variable `x`, which comes from the `for` loop clause, as well as the auxiliary variables `lst` and `rest`. The `in-list` macro is *hygienic* (Flatt 2016; Kohlbecker et al. 1986), so `lst` and `rest` are fresh variables that do not conflict with the rest of the program.

We call this intermediate representation an *expanded clause record* (ECR), and we say that Racket *expands* a `for` loop clause into an ECR. If the clause does not contain a fast sequence, it is translated to an ECR that computes the iteration methods at run time. We mainly use the record notation above for readability, but Racket actually represents ECRs with an S-expression syntax, and the syntax contains additional fields. We discuss the actual syntax in Section 3.1.

Now we can consider how to implement `fast-sequence-map`. Let us modify the previous example to use `fast-sequence-map` to square each element:

```
> (for ([y (fast-sequence-map sqr (in-list '(1 2 3)))])
    (printf "~a " y))
1 4 9
```

The `fast-sequence-map` form must produce an ECR. Its input in this example is the clause `[(y) (fast-sequence-map sqr (in-list '(1 2 3)))]`.[2] The components are the iteration variable `y`, the function `sqr`, and the sequence expression `(in-list '(1 2 3))`. The `fast-sequence-map` macro should bind the iteration variable `y` to the results of appying the function `sqr` to each element of the input sequence. The language of sequence expressions is extensible, so to handle an arbitrary input sequence expression we must first normalize it to a known, fixed structure—an ECR. That is, the essence of `fast-sequence-map` is a transformation from ECR to ECR.

In order to get the input ECR, however, we must first wrap the input sequence expression in a `for` loop clause, so we need an iteration variable to bind to its elements. We cannot use `y`, since `y` should be bound to elements of the result sequence. Instead, we generate a fresh variable `temp`.[3] Then we use the compile-time `expand-for-clause` function provided by the `for` DSL to expand the clause `[(temp) (in-list '(1 2 3))]`. The resulting ECR has the same form as the `in-list` ECR above but with `x` replaced with `temp`.

[2]Sequence macros receive the entire loop clause, unlike ordinary macros, which only receive their expression.

[3]Using a single generated variable `temp` means that `fast-sequence-map` only handles input sequences with single-valued elements (such as `in-list` but not `in-hash`). Unfortunately, Racket does not support querying a sequence macro to determine how many values each element has. We revisit this limitation in Section 4.

Now, we would like to simply add another inner binding for the squared value, like this:

```
loop-bindings  = ⟨ lst ← '(1 2 3) ⟩
pos-guard      = (pair? lst)
inner-bindings = ⟨ (temp) ← (car lst),
                  (rest) ← (cdr lst),
                  (y) ← (sqr temp) ⟩
loop-args      = ⟨ rest ⟩
```

Unfortunately, the right-hand sides of the inner bindings are not in the scope of previous inner bindings, because the loop skeleton uses `let-values` rather than `let*-values`, so this solution is not correctly scoped. We can fix the scope by combining the bindings into one that binds all of the inner variables and by using nested `let-values`, as follows:

```
inner-bindings = ⟨ (temp rest y) ← (let-values ([(temp) (car lst)]
                                                [(rest) (cdr lst)])
                                     (let-values ([(y) (sqr temp)])
                                       (values tmp rest y))) ⟩
```

Thus the expansion of the `for` loop for this example is the following:

```
(let loop ([lst '(1 2 3)])
  (when (pair? lst)
    (let-values ([(temp rest y)
                  (let-values ([(temp) (car lst)]
                               [(rest) (cdr lst)])
                                      ; The function application
                                      ; added by fast-sequence-map
                    (let-values ([(y) (sqr temp)])
                      (values temp rest x)))])
      (printf "~a " y)
      (loop rest))))
```

In summary, `fast-sequence-map` is essentially a transformation that adds an inner binding to its input ECR, with some additional rearrangement to satisfy the loop skeleton's scoping rules.

2.2 fast-sequence-filter

The `fast-sequence-filter` operation is more complicated. Let us discuss its expected behavior first. Consider the following example:

```
> (for ([x (in-list '(a b c d e))]
        [y (fast-sequence-filter odd?
             (in-list '(1 2 3 4 5 6)))])
    (printf "~v " (list x y)))
'(a 1) '(b 3) '(c 5)
```

The `fast-sequence-filter` sequence should bind `y` to only those elements of its input sequence that satisfy the given predicate. More precisely, it should satisfy two principles. First, it should be consistent with eager filtering (i.e., with the use of `filter` for lists) of its input sequence when it doesn't have side effects, like in this example. Second, it should not inspect its input sequence more than necessary, because other sequences might have side effects. That is, the first principle tells us what results it should produce, and the second principle tells us that it must produce them lazily—it cannot prepare them in advance.

On the first iteration of the `for` loop above, we look at the first sequence and extract its first element `'a`. Then, we look at the second sequence: the first element of its input sequence, `1`, satisfies the predicate `odd?`, so we bind `y` to it. On the second iteration, `x` gets bound to `'b`, and we look at the next candidate element for `y`, the value `2`. However, it doesn't satisfy the predicate, and that means that we should continue searching the second input sequence. Its next element is `3`, which satisfies the predicate, so we produce it as the next element for `y`. Note that *two* iterations of the `for` loop required *three* iterations of the `fast-sequence-filter`'s input sequence.

This example shows that `fast-sequence-filter` must use an inner loop to search for its next element. Furthermore, it cannot determine whether the result sequence is done without performing the search for the next element; after all, it is possible that its input sequence is not empty but doesn't contain any "good" values, and according to the laziness principle, we shouldn't inspect the input sequence until the next element is requested. Keeping these observations in mind will help us to figure out how to fit this into a loop skeleton.

In the previous subsection, we showed a simplified version of the loop skeleton and ECR, but now we need to use one of the components we omitted. The "pre-guard" represents another point where we can decide whether to stop looping. The pre-guard is checked after the inner-bindings are computed but before evaluating the loop body. We can use it as follows:

```
(let loop ([lst1 '(a b c d e)]
           [lst2 '(1 2 3 4 5 6)])
  (when (and (pair? lst1) #t) ; pos-guard
    (let-values ; inner-bindings
        ([(x) (car lst1)]
         [(rest1) (cdr lst1)]
         ; find the next y and rest2, or else y is done
         [(y-is-found y rest2) ____])
      (when (and #t y-is-found) ; pre-guard
        (printf "~v " (list x y))
        (loop rest1 rest2)))))
```

This code shows how Racket translates `for` loops with multiple clauses to recursive functions: the ECR binding components are concatenated and the ECR guard components are combined using `and`. In the position guard, `#t` for the second sequence means that we defer deciding whether `y` is done because we don't know that yet. In the inner bindings, we attempt to find the next element for the second sequence, and success or failure determines whether `y` is done.

To find the values for `y-is-found`, `y`, and `rest2`, we fill in the blank above with an inner loop over the second sequence, called `find-y-loop`. This inner loop may examine and reject elements from the input sequence before arriving at one of two possible results: it can find the next `y` (and the `rest` of the input sequence) or it may happen that there are no more elements—that is, `y` is done. Conceptually, the inner loop's result is an option of a pair, but we flatten it to three values to fit within the structure imposed by the loop skeleton. The `y-is-found` flag is used in the pre-guard to determine whether to continue.

The implementation of `fast-sequence-filter`, as with `fast-sequence-map`, creates a fresh iteration variable, synthesizes a clause, and expands it to get the input ECR. The result ECR's has the same loop variables and initialization expressions as the input ECR. Its inner bindings use the search loop to bind the found flag, the given iteration variable, and the next state of the input ECR's loop variables. The search loop (`find-y-loop`) is implemented using the input ECR's traversal components. The result ECR for this example is the following:

```
loop-bindings = ⟨ lst2 ← '(1 2 3 4 5 6) ⟩
pos-guard     = #t
inner-bindings = ⟨ (y-is-found y rest2) ←
                  (let find-y-loop ([lst lst2]) ; inner search loop
                    (cond [(pair? lst)
                           (let ([temp (car lst)]
                                 [rest (cdr lst)])
                             (cond ; the filtering predicate
                               [(odd? temp) (values #t temp rest)]
                               [else (find-y-loop rest)]))]
                          [else (values #f #f #f)])) ⟩
pre-guard     = y-is-found
loop-args     = ⟨ rest2 ⟩
```

This transformation effectively *reinterprets* an intermediate representation intended to be used with the `for` loops, effectively introducing a new loop skeleton. We must ensure that the new interpretation is compatible. We return to this issue in Section 3.3.

3 IMPLEMENTATION AND CORRECTNESS

In the previous section, we introduced ECRs as abstract compile-time records and described sequence macros as transformations on these abstract records. In this section, we show the concrete implementations in terms of Racket's macro system. First, we specify the S-expression syntax that Racket uses to represent ECRs, then we show the implementation of the transformations using Racket's pattern-matching and template macro toolkits.

Beyond the simple syntactic structure, however, `for` loop clauses and ECRs also have binding structure, and our transformations must respect that structure. So we add binding information to the specification of ECRs and to the macro syntax patterns, and we annotate the macro templates with arguments for their scope-correctness. Racket's macro system does not directly support such information, so we express it informally using a type-like notation in comments.

3.1 Representing ECRs

Before we discuss ECR representations, we must introduce one more field: the "outer bindings".[4] The outer bindings are bound before starting iteration in an extra `let`-expression that wraps the loop. They are commonly used to pre-compute values used in iteration and to avoid repeatedly evaluating argument expressions. For example, the expansion of `in-range` would actually begin thus:

```
[(n) (in-range start end step)]  ⇒   outer-bindings = ⟨ start-var ← start,
                                                        end-var ← end,
                                                        step-var ← step ⟩
                                     ....
```

Here, `start-var`, `end-var`, and `step-var` are freshly generated variable names.

Here is the `for` loop skeleton updated to include the outer bindings:

```
(let ([outer-id outer-expr])
  (let loop ([loop-id loop-expr] ...)
```

[4] In fact, Racket's actual ECR syntax contains two other fields: an "outer check" expression between the outer bindings and the beginning of the loop, and a "post-guard" expression after evaluating the loop body but before continuing to the next iteration. We don't use either in this paper, and this version of ECR is the last version that we show.

```
(when pos-guard
  (let-values ([(inner-id ...) inner-expr] ...)
    (when pre-guard
      loop-body
      (loop loop-arg ...))))))
```

Racket represents ECRs using S-expression syntax consisting of a list of the components. For instance, the example clause `[(x) (in-list '(1 2 3))]` from Section 2.1 translates to an ECR represented by the following syntax:

```
(([(lst) '(1 2 3)])   ; outer bindings
 ([lst lst])          ; loop bindings
 (pair? lst)          ; position guard
 ([(x) (car lst)]     ; inner bindings
  [(rest) (cdr lst)])
 #t                   ; pre-guard
 (rest))              ; loop arguments
```

The syntactic structure of ECRs can be described by the following *syntax class* (Culpepper 2012):

```
(begin-for-syntax
  ;; Represents an expanded clause record
  (define-syntax-class ECR
    (pattern
     [([(outer-id:id ...) outer-rhs:expr] ...)
      ([loop-id:id loop-expr:expr] ...)
      pos-guard:expr
      ([(inner-id:id ...) inner-rhs:expr] ...)
      pre-guard:expr
      (loop-arg:expr ...)]
     )))
```

The pattern shows that an ECR consists of six components and it specifies the syntactic structure of each component. The pattern variables are annotated with syntax classes; for example, `outer-id:id` says that each `outer-id` must be an identifier (`id`), and `pos-guard:expr` says that the `pos-guard` is an expression. Ellipses indicate repetition.

In addition to the syntactic structure, though, we also want to specify the binding structure. And we may as well generalize from binding structure to type structure. We can think of scope errors as type errors: for example, a reference to an unbound variable has no type; a wrongly captured reference has the type of the wrong binder rather than the type of the intended one. Although Racket doesn't have a built-in notion of types, Racket programmers commonly impose ad hoc type disciplines on the programs that they write. So let us specify those types informally at the level of code comments and use them to reason about macros.

Figure 1 extends the previous syntax class definition with comments that specify the type and binding structure of the ECR components. Each pattern variable that stands for an expression is followed by a comment using the `#|...|#` notation with the pattern variable's *shape*. The shape *Expr*`[`Γ`][`τ`]` means that the variable stands for an expression that is evaluated in an environment compatible with the type environment Γ and the expression has type τ. That is, if `e` has shape *Expr*`[`Γ`][`τ`]`, that corresponds to the type judgment $\Gamma \vdash \texttt{e} : \tau$.

```
(begin-for-syntax
  ;; Represents an expanded clause record
  (define-syntax-class ECR  #|ECR[Γ][Δ]|#
    (pattern
     [([(outer-id:id ...) outer-rhs:expr #|Expr[Γ][(values τO ...)]|#] ...)
      ([loop-id:id loop-expr:expr #|Expr[Γ/ΔO][τL]|#] ...)
      pos-guard:expr #|Expr[Γ/ΔO/ΔL][Boolean]|#
      ([(inner-id:id ...) inner-rhs:expr #|Expr[Γ/ΔO/ΔL]
                                           [(values τI ...)]|#] ...)
      pre-guard:expr #|Expr[Γ/ΔO/ΔL/ΔI][Boolean]|#
      (loop-arg:expr #|Expr[Γ/ΔO/ΔL/ΔI][τL]|# ...)]
     ;; where ΔO = {outer-id:τO ... ...}
     ;;       ΔL = {loop-id:τL ...}
     ;;       ΔI = {inner-id:τI ... ...}
     ;; and   Δ  ⊆ ΔO/ΔL/ΔI
     )))
```

Fig. 1. Specification of ECR

The shape of an ECR pattern variable is determined by its usage in the `for` loop skeleton. Consider the shape of `loop-expr`, *Expr*`[Γ/ΔO][τL]`. The loop expression `loop-expr` must be in the scope of the initial inherited type environment Γ and it also must be in the scope of the outer bindings. We use the abbreviation ΔO to describe the type environment "delta" corresponding to the outer bindings. We say that ΔO is an "environment rib", and we write Γ/ΔO to mean that ΔO is bound after the bindings from Γ0 and thus can shadow them. We adopt the convention of using Γ to refer to complete type environments and Δ to refer to partial type environments.

In other words, a shape of a pattern tells us how that pattern is supposed to be used in a macro template. In a macro, we might want to take some patterns apart and use their subpatterns in a different place, but we must put them in the same scope in which they are defined.

The `ECR` syntax class also defines a shape, with two parameters: the initial type environment `Γ0` and the type environment delta Δ for the iteration variables that it binds. It is not apparent from an ECR's contents which bindings in an ECR correspond to the iteration variables; the iteration variables are determined by the `for` clause that produced the ECR. The iteration variables (Δ) are significant for ECRs, though, because we assume and require that every identifier in dom(ΔO/ΔL/ΔI) − dom(Δ) is fresh.

Likewise, we can specify the shape of `for` clauses as follows:

```
(begin-for-syntax
  (define ForClause  #|ForClause[Γ][{iter-var:τ ...}]|#
    (pattern [(iter-var:id ...) seq:expr #|Sequence[Γ][(values τ ...)]|#])))
```

As with `ECR`, the first parameter (Γ) represents the inherited type environment and the second parameter represents the iteration variables and their types. The shape *Sequence*`[Γ][(values τ ...)]` represents sequences whose elements consist of as many values as there are types τ, where each value has the correspond type τ. For example, `(in-range 0 5)` has shape *Sequence*`[Γ0][Nat]`, and `(in-hash (hash 'a 1 'b 2))` has shape *Sequence*`[Γ0][(values Symbol Nat)]`. Any expression that has a sequence type also matches the *Sequence* shape. That is:

$$\textit{Expr}\texttt{[Γ][(sequenceof (values τ ...))]} \subset \textit{Sequence}\texttt{[Γ][(values τ ...)]}$$

Here are some example `for` clauses and their shapes in the initial environment `Γ0`:

```
[(x)   (in-list '(1 2 3))]    ; ForClause[Γ0][{x:Nat}]
[(k v) (in-hash (hash 'a 1))] ; ForClause[Γ0][{k:Symbol,v:Nat}]
```

The expansion of a *ForClause* to an *ECR* preserves its parameters:

$$\mathit{ForClause}[\Gamma][\Delta] \quad \Rightarrow \quad \mathit{ECR}[\Gamma][\Delta]$$

Our sequence macros use the `ECR` syntax class to parse and destructure intermediate ECRs. They do not use the `ForClause` syntax class, since their input patterns are specialized to the variant of the *Sequence* shape that they implement. The following grammar summarizes their type and binding structure:

$\mathit{Sequence}[\Gamma][\tau]$ ::= ....
| (fast-sequence-map $\mathit{Expr}[\Gamma][\tau' \rightarrow \tau]$ $\mathit{Sequence}[\Gamma][\tau]$)
| (fast-sequence-filter $\mathit{Expr}[\Gamma][\tau \rightarrow \texttt{Boolean}]$ $\mathit{Sequence}[\Gamma][\tau]$)

3.2 Implementing fast-sequence-map

Now our goal is to show the implementation of `fast-sequence-map` and check its correctness in connection with the specification of ECR by using our type notation in comments.

The implementation follows the same basic structure outlined in the previous section. As a first step, it introduces a temporary iteration variable and expands a clause containing that variable and the original sequence expression to get an input ECR. And as a second step, it transforms that ECR to bind the original iteration variables to the result of applying the given function. The transformation uses components of the input ECR, and we must ensure that we use them in the correct scope. And thus the new part that we are going to focus on is checking the binding structure of ECR specified in its representation, for which we use code comments with the type information.

We implement `fast-sequence-map` using the `define-sequence-syntax` form provided by Racket's API for defining sequence macros. In the code shown in Figure 2, line 2 specifies that, if a `fast-sequence-map` expression is not used directly in a `for` loop clause, the dynamic function `sequence-map` will be called instead.

On lines 5 through 7, `syntax-parse` matches the `for` loop clause against the pattern

```
[(iter-var:id ...) (fast-sequence-map
                     f:expr   #|Expr[Γ0][τ -> (values τ' ...)]|#
                     seq:expr #|Sequence[Γ0][τ]|#)]
```

The comments specify how we interpret all the input pattern variables. Namely, `seq` gets interpeted as a sequence of single-valued elements of type τ and `f` as a function accepting an argument of type τ and returning values of types τ' ... that become the types of the output sequence's elements, as specified by the *ForClause* shape.

The first step, obtaining the input ECR, is performed by introducing a new variable `temp-id` for the input sequence expression `seq` and expanding the clause `[(temp-id) seq]` using `expand-for-clause`. (Note that the code in Figure 2 introduces other new variables, `f*` and `ok`. Those variables are fresh because of Racket's macro hygiene mechanism. However, for generating the `temp-id` identifier, we use the `generate-temporaries` function from Racket's API, as it is required by `expand-for-clause` to ensure that it is fresh.) The `expand-for-clause` function accepts two syntax objects. The first one is the original syntax, which is only needed for reporting syntax errors. The second one is the `for` loop clause to be expanded. Hence, the `for` clause is wrapped in the `#'` quote that accepts a template and constucts a syntax object. Lines 10 and 11 check that the result of the expansion matches the pattern of the `ECR` syntax class and adds pattern variables for its components with names `ecr.outer-id`, `ecr.outer-rhs`, etc. to the scope of the macro template

```
1 (define-sequence-syntax fast-sequence-map
2   (lambda () #'sequence-map)
3   (lambda (stx)
4     (syntax-parse stx  #|ForClause[Γ0][{iter-var:τ' ...}]|#
5       [[(iter-var:id ...) (fast-sequence-map
6                            f:expr   #|Expr[Γ0][τ -> (values τ' ...)]|#
7                            seq:expr #|Sequence[Γ0][τ]|#)]
8        ;; ==>
9        #:with (temp-id) (generate-temporaries #'(temp-id))
10       #:with ecr:ECR #|ECR[Γ0][{temp-id:τ}]|#
11              (expand-for-clause stx #'[(temp-id) seq])
12       ;; result  #|ForClause[Γ0][{iter-var:τ' ...}]|#
13       #'[(iter-var ...)
14          (:do-in  #|ECR[Γ0][{iter-var:τ' ...}]|#
15           ;; outer bindings
16           ([(ecr.outer-id ...) ecr.outer-rhs] ...
17            [(f*) f])
18           ;; loop bindings
19           ([ecr.loop-id ecr.loop-expr] ...)
20           ;; pos check
21           ecr.pos-guard
22           ;; inner bindings
23           ([(ecr.inner-id ... ... iter-var ... ok)
24             (let-values ([(ecr.inner-id ...) ecr.inner-rhs] ...)
25               (cond
26                 [ecr.pre-guard
27                  (let-values ([(iter-var ...) (f* temp-id)])
28                    (values ecr.inner-id ... ... iter-var ... #t))]
29                 [else
30                  (let ([ecr.inner-id #f] ... ... [iter-var #f] ...)
31                    (values ecr.inner-id ... ... iter-var ... #f))]))])
32           ;; pre-guard
33           ok
34           ;; loop args
35           (ecr.loop-arg ...))]]
36       [_ #f])))
```

Fig. 2. Implementation of `fast-sequence-map`

that starts on line 14. The annotation on line 10 says that the `ecr` components have the shapes specified by *ECR*`[Γ0][{temp-id:`τ`}]`.

The result of the `fast-sequence-map` expansion starting on line 12 is a new clause with shape *ForClause*`[Γ0][{iter-var:`τ`' ...}]` constructed by wrapping the output ECR in the special `:do-in` syntax. A `for` loop and its variants interpret a `:do-in` form by splicing parts of the ECR it wraps into the loop skeleton. The comment on line 14 indicates that the syntax following `:do-in` must be an *ECR*`[Γ0][{iter-var:`τ`' ...}]`.

Let us now focus on parts of the result ECR to check their correctness using our type notation. The result ECR uses components of the input ECR, and the only fields that it changes are the outer bindings, inner bindings, and pre-guard. The annotation on line 10 specifies the shapes of the input ECR components. The annotation on line 14 specifies the shape that the template's result ECR will be interpreted according to. When we use an input ECR component in the result ECR, we must make sure it is used in a way compatible with its original specification.

Consider the outer bindings first. In the input ECR `ecr`, the outer bindings were (where i in τ`i0` stands for "input"):

```
([(ecr.outer-id ...) ecr.outer-rhs #|Expr[Γ0][(values τi0 ...)]|#] ...)
```

The `fast-sequence-map` macro transforms it to (where o stands for "output"):

```
;; Should be [(Id ...) Expr[Γ0][(values τo0 ...)]] ...
;; It matches, where τo0 ... ... = τi0 ... ... (τ -> (values τ' ...))
([(ecr.outer-id ...) ecr.outer-rhs] ...
 [(f*) f])
```

Here, we bind a fresh identifier `f*` to the input function `f` at the outer bindings that evaluate only once in order to prevent it from re-evaluating on each iteration of the loop. This code is correct because it matches the shape of outer bindings specified by ECR and it uses components of the input ECR in the right scope—that is, they don't depend on anything other than bindings from Γ0. We show that in the comments.

Now, consider the inner bindings. They must be a sequence of clauses whose left-hand side is a sequence of identifiers.

```
;; Should be [(Id ...) Expr[Γ0/Δo0/ΔoL][(values τoI ...)]] ...
([(ecr.inner-id ... ... iter-var ... ok)  ;; Id ... ... Id ... Id
                                          ;; It matches.
  (let-values ....)])
```

The code in Figure 2 regroups the shape

```
;; [(Id ...) Expr[Γ0/Δo0/ΔoL][(values τoI ...)]] ...
```

as

```
;; [(Id ... ...) Expr[Γ0/Δo0/ΔoL][(values τoI ... ...)]]
```

and also adds more identifiers to the left-hand side sequence, but that is allowed, as these shapes match. There is a new identifier `ok`, not present in the previous section, where we showed a simpler version of `fast-sequence-map`. We bind `ok`, an analog of `y-is-found` from `fast-sequence-filter`, to prevent re-evaluating the pre-guard, which we use to check whether the current element found in the input sequence is "good."

As in the previous section, `fast-sequence-map` applies the given function to the temporary iteration variable, bound to the values of the input sequence's element on the current iteration. The temporary variable `temp-id` is part of the input ECR's bindings and must be in the scope `Γ0/Δi0/ΔiL/ΔiI`. Thus, next, we bind the inner bindings of the input ECR in the `let-values` expression:

```
(let-values
    ([(ecr.inner-id ...)
      ;; Should be:      Expr[Γ0/Δi0/ΔiL][τiI] ...
      ;; Type checks as: Expr[Γ0/Δi0,f*:(τ -> (values τ' ...))/ΔiL][τiI] ...
      ;; f* is fresh, so the scope is OK.
```

```
      ecr.inner-rhs] ...)
  ....)
```

The `ecr.inner-rhs` component had type *Expr*`[Γ0/ΔiO/ΔiL][τiI] ...` in the input ECR. When we use it in the output ECR, we must ensure that we put it in the correct scope. However, when we type check it, we can see that it is used in the scope that has one more binding: `f*:(τ -> (values τ' ...))`. But this is actually fine because the `f*` variable is fresh, and thus doesn't affect any expressions that appear in its scope. The same reasoning applies to all the components that occur in the outer ECR after the outer bindings, and hence have `f*` in their scopes.

The function should be applied to the current element of the input sequence only if it is a "good" element of that sequence according to the `ecr.pre-guard` component. Let us focus on it. If it is not a true value, then the found element is not considered as an element of the input sequence—that's the high-level meaning of a pre-guard. And thus it must not be considered in the output sequence. That means that here we should use the input sequence's pre-guard at the "inner bindings" field of the output ECR to check its value. That is checked by the `cond` expression:

```
(cond
  ;; Expr[Γ0/ΔiO,f*:(τ -> (values τ' ...))/ΔiL/ΔiI][Boolean]
  ;; The scope is OK.
  [ecr.pre-guard
   (let-values ([(iter-var ...) (f* temp-id)])
     (values ecr.inner-id ... ... iter-var ... #t))]
  [else
   (let ([ecr.inner-id #f] ... ... [iter-var #f] ...)
     (values ecr.inner-id ... ... iter-var ... #f))])
```

Again, when we use `ecr.pre-guard`, we must ensure that we put it in the right scope, having inner bindings of the input ECR. If we inspect the bindings in whose scope it occurs in the output ECR, we find that the scopes match.

When `ecr.pre-guard` is a true value, we apply the function to `temp-id`, which has type τ and is part of `ΔiO/ΔiL/ΔiI`, and return the iteration variable `temp-id ...` bound to the result as part of the output ECR's inner bindings. The type checking process for this is similar to the previously shown, and we omit the comments. Otherwise, if `ecr.pre-guard` is false, that means that `temp-id` is not a valid value, and since we cannot apply the function, we return the appropriate number of `#f` values. And as this expression doesn't have any free variables, we can safely put it in the scope `Γ0/ΔoO/ΔoL/ΔoI` due to the standard weakening rule.

The whole `cond` expression thus has the following shape, which matches the specification of ECR:

```
;; (cond ....) :: Expr[Γ0/ΔoO/ΔoL][(values τoI ...)]]
;; where ΔoO = ΔiO,f*:(τ -> (values τ' ...)); ΔoL = ΔiL
;;       τoI ... ... = (U τiI #f) ... ... (U τ' #f) ... Boolean
```

Finally, the last component that we change in the input ECR is pre-guard:

```
;; Should be Expr[Γ0/ΔoO/ΔoL/ΔoI][Boolean]
;; ΔoI = ΔiI,iter-var:(U τ' #f)...,ok:Boolean
;; Γ0/ΔoO/ΔoL/ΔoI maps ok to Boolean, so it matches.
ok
```

The ok variable is bound to #t when the new element of the output sequence is successfully evaluated, and to #f otherwise. The type of the freshly generated variable ok is *Expr*[∅][Boolean], but due to the weakening rule it is OK to interpret it as *Expr*[Γ0/ΔoO/ΔoL/ΔoI][Boolean].

3.3 Implementing fast-sequence-filter

We can apply the similar reasoning as before to check the scoping of fast-sequence-filter. The code shown in Figure 3 again in general follows the structure from the previous section.

Let us focus on the inner bindings. Consider the let loop expression on line 23:

```
                          ;; ecr.loop-id :: Expr[ΔiL][τiL] ...
(let loop ([ecr.loop-id ecr.loop-id] ...)
;; Let Δother-stuff = loop:(τoL ... -> (values τoI ...))/loop-id:τoL...
;; where toL = ....
  ....)
```

This code is a bit different from the version shown in the previous section: here we use the ecr.loop-id variables, as other components depend on their values in the inner loop. We check that loop-ids are used in the correct scope, as they are identifiers that don't depend on any variables that are not in ΔiL. We will use the Δother-stuff abbreviation for describing the new bindings that the let loop expression introduces in the scope of its body.

Now, ecr.pos-guard in its body on line 26 type checks as follows:

```
;; ecr.pos-guard :: Expr[Γ0/ΔiO/ΔiL][Boolean]
;; And it's OK to interpret it as Expr[Γ0/ΔiO/ΔiL/Δother-stuff][Boolean]
;; because each variable in Δother-stuff is either fresh
;; or in ΔiL and has the proper type
(if ecr.pos-guard ....)
```

We put ecr.pos-guard from the input ECR in the "inner bindings" field of the output ECR, with the new scope additionally including Δother-stuff. But it is fine because Δother-stuff contains only the variable loop that is fresh and ecr.loop-id ... that are present in ΔiL with the same type. The rest of the scope matches.

Also, line 26 binds fresh variables loop-arg* ... to ecr.loop-args in order to avoid copying those expressions:

```
;; ecr.loop-arg :: Expr[Γ0/ΔiO/ΔiL/ΔiI][τiL]
;; Used as Expr[Γ0/ΔiO/ΔiL/Δother-stuff/ΔiI][τiL]
(let ([loop-arg* ecr.loop-arg] ...) ....)
```

The loop-arg* variables are generated by the generate-temporaries function on line 8 to be of the same length as the ecr.loop-arg ... sequence. Again, here we use loop arguments from the input ECR in the inner bindings of the output ECR. If we inspect their scope, it matches, where the same reasoning as before applies to Δother-stuff.

Checking the remaining components is quite similar.

4 FROM COMPREHENSION TO ITERATION

We've outlined the solution for filtering and mapping, two common and fundamental sequence operations. Now let us consider what they cannot express and use that to build a general tool for composing fast sequences.

```
(define-sequence-syntax fast-sequence-filter
  (lambda () #'sequence-filter)
  (lambda (stx)
    (syntax-parse stx
      [[(iter-var:id ...) (_ f seq:expr)]
       #:with ecr:ECR (expand-for-clause stx #'[(iter-var ...) seq])
       #:with (loop-id* ...) (generate-temporaries #'(ecr.loop-id ...))
       #:with (loop-arg* ...) (generate-temporaries #'(ecr.loop-arg ...))
       #:with (false* ...)
              (map (lambda (x) #'#f)
                   (syntax->list #'(ecr.inner-id ... ... loop-arg* ...)))
       #'[(iter-var ...)
          (:do-in
           ;; outer bindings
           ([(ecr.outer-id ...) ecr.outer-rhs] ...
            [(f*) f])
           ;; loop bindings
           ([ecr.loop-id ecr.loop-expr] ...)
           ;; pos check
           #t
           ;; inner bindings
           ([(ecr.inner-id ... ... loop-id* ... ok)
             (let loop ([ecr.loop-id ecr.loop-id] ...)
               (if ecr.pos-guard
                   (let-values ([(ecr.inner-id ...) ecr.inner-rhs] ...)
                     (let ([loop-arg* ecr.loop-arg] ...)
                       (if ecr.pre-guard
                           (if (f* iter-var ...)
                               (values ecr.inner-id ... ...
                                       loop-arg* ... #t)
                               (loop loop-arg* ...))
                           (values false* ... #f))))
                   (values false* ... #f)))])
           ;; pre guard
           ok
           ;; loop args
           (loop-id* ...))]]
      [_ #f])))
```

Fig. 3. Implementation of `fast-sequence-filter`

4.1 Nesting

As a new motivating example, suppose we have the following structure for a set:

```
; (Setof X) = (custom-set (Vectorof (Listof X)))
> (struct custom-set (table))
```

The structure contains a vector of lists, where some hash function determines the index in the vector where an element is stored, and each vector slot contains a list of elements. For example:

```
; {1, 7, 5, 8, 4} could be represented by the following:
> (define s (custom-set (vector (list 1 7 5) (list) (list) (list 8 4))))
```

Now, suppose we want to print the elements of the set `s`. One way is the following:

```
> (for ([x (in-vector (custom-set-table s))] ; a binding clause
        #:when (pair? x) ; a when-clause
        [y (in-list x)])
    (printf "~a " y))
1 7 5 8 4
```

The presence of the `#:when` clause between the two binding clauses causes the second clause's iteration to be nested within the first clause's iteration, rather than the two clauses iterating in parallel. Additionally, the `#:when` condition expression and the second clause's sequence expression are in the scope of the first clause's iteration variable.[5]

The solution above requires the writer of the `for` loop to know the representation of these sets and to have access to the structure's accessor. It would be better to provide a reusable sequence form, `in-custom-set`, that encapsulated the structure access and the nested iteration. The sequence `(in-custom-set s)` should have the elements `1`, `7`, `5`, `8`, `4`. For that goal, we have designed a sequence composition macro named `in-nested`. The `in-nested` macro "flattens" the nested iteration space into a sequence as it is shown below:

```
> (define-sequence-rule (in-custom-set s)
    (in-nested ([x (in-vector (custom-set-table s))])
      (in-list x)))
> (for ([x (in-custom-set s)])
    (printf "~a " x))
1 7 5 8 4
```

The syntax and type structure of `in-nested` are summarized by the following grammar:

Sequence[Γ][τ] ::= ....
| (in-nested (*ForClause*[Γ][Δ] ...) *Sequence*[Γ/Δ...][τ])

4.2 Implementing in-nested

The `in-nested` form actually provides both parallel iteration and nested iteration. For example:

```
> (for ([x (in-nested ([y (in-list '(a b c d e))] ; merge
                       [z (in-range 1 6 2)])      ;
            (in-list (list y z)))])
    (printf "~a " x))
a 1 b 3 c 5
```

That is, the group of "outer" `for` clauses are iterated in parallel, and for each step of the outer iteration, the "inner" sequence is completely iterated through.

We will discuss the implementation of parallel and nested iteration separately.

[5]In this example, the condition is redundant; skipping the iteration of an empty `x` has no effect. We could replace the clause with `#:when #t`, but we cannot delete the `#:when` clause entirely, since its presence separates the nesting levels.

4.2.1 Nested Iteration. Let us first solve the simpler problem of implementing `in-nested` without support for parallel iteration. That is, our first version of `in-nested` has the following syntax:

Sequence[Γ][τ] ::= ....
| (in-nested (*ForClause*[Γ][Δ]) *Sequence*[Γ/Δ][τ])

For the sake of illustration, consider the case where the *outer sequence* is a list of numbers, and the *inner sequence* is the range of natural numbers less than each outer element:

```
> (for ([i (in-nested ([n (in-list '(3 4 5))]) ; outer
              (in-range 0 n))]) ; inner
    (printf "~a " i))
0 1 2 0 1 2 3 0 1 2 3 4
```

We can write this as a recursive function according to the following strategy:

Case 1: Look into the inner sequence and get the next element, or
Case 2: if the inner sequence is empty, look into the outer sequence to get the next inner sequence and try again, or
Case 3: if the outer sequence is empty, then stop.

The strategy depends on two input ECRs. The outer ECR comes from expanding the outer binding clause. The inner ECR comes from expanding the clause `[(i) (in-range 0 n)]`; it uses the main loop's iteration variable because that variable receives the values produced by the inner sequence. We have already shown ECRs for `in-list`. An `in-range` clause expands as follows:

```
[(i) (in-range start end step)]  ⇒  loop-bindings = ⟨ pos ← start ⟩
                                    pos-guard     = (< pos end)
                                    inner-bindings = ⟨ (i) ← pos ⟩
                                    pre-guard     = #t
                                    loop-args     = ⟨ (+ pos step) ⟩
```

Figure 4 shows a translation of our example of `in-nested` above into a recursive function. For readability, we have prefixed the outer ECR's auxiliary variables with `outer-` (for example, `outer-lst` instead of `lst`), and likewise for the inner ECR.

The resulting loop arguments carry the state of the outer sequence (`outer-lst`), the current element of the outer sequence (`n`), and the state of the inner sequence (`inner-pos`). The current outer element is needed because the inner ECR might depend on it; for example, see the check `(< inner-pos n)` for Case 1. The inner sequence state starts uninitialized, due to the laziness principle; we consider that case the same as the current inner sequence being empty. When we recur, after the body, we advance the state of the inner sequence.

As with `fast-sequence-filter`, it is impossible to tell whether the sequence is exhausted without looking for the next element, so we add a flag `i-is-found?` to the inner bindings to indicate whether another element is available. The `nested-loop` function attempts to find the next element according the the strategy above. The Case 1 code uses the inner ECR's inner bindings to produce a new element. The Case 2 code uses the outer ECR's components to produce an outer element, and the outer iteration variable (`n`), and advance the outer sequence state; then it initializes a new inner sequence and retries with the new state. The Case 3 code signals that no element was found.

As with `fast-sequence-filter`, we are reinterpreting the components of the input ECRs with a new loop skeleton. In addition, we must respect the binding structure of `in-nested`: the inner sequence expression is in the scope of the outer iteration variables.

```
(let loop (; outer sequence state:
           [outer-lst '(3 4 5)]
           ; inner sequence state:
           [inner-initialized? #f]
           [n uninitialized]
           [inner-pos uninitialized])
  (when #t
    (let-values
        ([(i-is-found? i    ; current iteration results
           outer-lst        ; outer sequence state
           n inner-pos)     ; inner sequence state
          (let nested-loop ([outer-lst outer-lst]
                            [inner-initialized? inner-initialized?]
                            [n n]
                            [inner-pos inner-pos])
            (cond
              ; Case 1
              [(and inner-initialized?
                    (< inner-pos n))    ; use inner pos-guard
               (let ([i inner-pos])     ; use inner inner-bindings
                 (values #t i outer-lst n inner-pos))]
              ; Case 2
              [(pair? outer-lst)                ; use outer pos-guard
               (let ([n (car temp-outer-seq)]   ; use outer inner-bindings
                     [outer-rest (cdr outer-rest)])
                 (let ([next-outer-lst outer-rest]) ; use outer loop-args
                   (let ([i 0])                 ; use inner loop-bindings
                     (nested-loop next-outer-lst #t n i))))]
              ; Case 3
              [else
               (values #f #f #f #f #f)]))])
      (when i-is-found?
        (printf "~a " i)    ; body
        (let ([next-inner-pos (+ inner-pos 1)]) ; use inner loop-args
          (loop outer-lst #t n next-inner-pos))))))
```

Fig. 4. Recursive function for `in-nested`

4.2.2 Parallel Iteration. The general version of `in-nested` supports multiple binding clauses. To handle that, we don't need to change the implementation of nesting. Instead, we expand the outer binding clauses of `in-nested` and *merge* the resulting ECRs into a single ECR. Then we apply nesting to the single outer ECR and the single inner ECR, as we described previously. Recall the example from the beginning of this section:

```
(for ([x (in-nested ([y (in-list '(a b c d e))]
                     [z (in-range 1 6 2)]) ; start, end, step
           (in-list (list y z)))])
```

```
  (printf "~a " x))
```

In this expression, we expand the two clauses `[(y) (in-list '(a b c d e))]` and `[(z) (in-range 1 6 2)]` and merge their ECRs together to the following ECR:

$$\begin{array}{lll} \textit{loop-bindings} & = & \langle\ \texttt{lst} \leftarrow \texttt{'(a b c)},\ \texttt{pos} \leftarrow \texttt{1}\ \rangle \\ \textit{pos-guard} & = & \texttt{(and (pair? lst) (< pos 6))} \\ \textit{inner-bindings} & = & \langle\ \texttt{(x)} \leftarrow \texttt{(car lst)},\ \texttt{(rest)} \leftarrow \texttt{(cdr lst)},\ \texttt{(y)} \leftarrow \texttt{pos}\ \rangle \\ \textit{pre-guard} & = & \texttt{(and \#t \#t)} \\ \textit{loop-args} & = & \langle\ \texttt{rest},\ \texttt{(+ pos 2)}\ \rangle \end{array}$$

We merge the ECRs by merging each component independently. For the lists *loop-bindings*, *inner-bindings*, and *loop-args*, we concatenate them for each ECR. And the merged *pos-guard* and *pre-guard* are conjunctions of the individual guards' expressions.

This naive form of merging changes the scopes that expressions occur in. In the following sections, we discuss the problems that arise and different ways of fixing them.

4.3 The Problem with Naive Merging

We've outlined our model of the `in-nested` implementation. However, as we already briefly mentioned before, its parts are actually more complicated. Now, we consider some less trivial examples and see what errors may rise in the naive solution.

A useful building block is the sequence form `(in-when condition)`, which evaluates the boolean condition expression and produces either one element of zero values or zero elements. Here is its specification:

$$\begin{array}{rl} \textit{Sequence}[\Gamma][\texttt{(values)}] & ::= \ \ldots. \\ & \mid\ \texttt{(in-when}\ \textit{Expr}[\Gamma][\texttt{(values)}]\texttt{)} \end{array}$$

And here is one possible implementation, although not the most straightforward one:

$$\texttt{[() (in-when condition-expr)]} \ \Rightarrow\ \begin{array}{lll} \textit{loop-bindings} & = & \langle\ \texttt{do-iter} \leftarrow \texttt{\#t}\ \rangle \\ \textit{pos-guard} & = & \texttt{do-iter} \\ \textit{pre-guard} & = & \texttt{condition-expr} \\ \textit{loop-args} & = & \langle\ \texttt{\#f}\ \rangle \end{array}$$

Consider the following example with `in-nested` and `in-when`:

```
> (define x 1)
> (for ([y (in-nested ([x (in-list (list (list x 2)))]
                       [() (in-when (odd? x))])
              (in-list x))])
    (println y))
odd?: contract violation
  expected: integer?
  given: '(1 2)
```

This example above has two binding clauses that must be iterated over in parallel. The `(odd? x)` expression in the second clause is supposed to refer to the global `x` bound to `1`. However, evaluating this expression raises an error saying that `odd?` is applied to the list `'(1 2)`. That is, the reference to `x` in the second clause wrongly gets captured by the first clause. If we rename the global definition of `x` and its intended reference to `z`, the error disappears:

```
> (define z 1)
> (for ([y (in-nested ([x (in-list (list (list x 2)))]
                       [() (in-when (odd? z))])
```

```
            (in-list x))])
    (println y))
1
2
```

Let us see why this happens. For the parallel iteration, our solution merges the ECRs component-wise to produce the following ECR:

$$
\begin{array}{lll}
\textit{loop-bindings} & = & \langle\ \texttt{lst} \leftarrow \texttt{(list (list x 2))}, \\
 & & \quad \texttt{do-iter} \leftarrow \texttt{\#t}\ \rangle \\
\textit{pos-guard} & = & \texttt{(and \#t do-iter)} \\
\textit{inner-bindings} & = & \langle\ \texttt{(x)} \leftarrow \texttt{(car lst)}, \\
 & & \quad \texttt{(rest)} \leftarrow \texttt{(cdr lst)}\ \rangle \\
\textit{pre-guard} & = & \texttt{(and \#t (odd? x))} \\
\textit{loop-args} & = & \langle\ \texttt{rest}, \\
 & & \quad \texttt{\#f}\ \rangle
\end{array}
$$

Let's compare the scope of the pre-guard `(odd? x)` in the original ECR versus the merged ECR. In the original ECR:

`(odd? x)` is in the scope of `do-iter` and the original environment

In the merged ECR, where it is a sub-expression of the pre-guard:

`(odd? x)` is in the scope of `x`, `rest`, `lst`, `do-iter`, and the original environment

That is, the merged ECR has put the `(odd? x)` expression in the scope of additional bindings from the other ECR that could capture its free variables. In fact, it is safe to add `lst` and `rest` to the environment of `(odd? x)`, because those auxiliary variables are fresh. But the addition of `x`, which is an iteration variable, captures the free reference in the expression and causes the error shown above.

4.4 One Fix for Merging

The problem with `merge` is that the expressions from one ECR are put into the scope of bindings from the other ECR. One solution is to lift the expressions to the outer bindings, wrap them with `lambda`s to recreate the expected environment, and replace their original occurrences with calls to the lifted functions.

For example, we would lift `a.pre-guard` to a fresh outer binding like this:

```
[(tmp-a-pre-guard) (lambda (a.outer-id ... ...)
                     (lambda (a.loop-id ...)
                       (lambda (a.inner-id ... ...)
                         a.pre-guard)))]
```

and then replace the use of `a.pre-guard` in the result with

```
(((tmp-a-pre-guard a.outer-id ... ...) a.loop-id ...) a.inner-id ... ...)
```

We can further optimize away the nested `lambda`s to a single `lambda` wrapper, and in many cases, the Racket compiler can inline away the new function. This is a common technique in macro writing for avoiding unwanted scopes.

Note that the replacement pre-guard does not rely on the base environment (Γ) at all; any such references were moved to the lifted function in the outer bindings. That means that the replacement expression satisfies a more restricted shape: *Expr*`[a.ΔO/a.ΔL/a.ΔI][Boolean]`—note the absence of Γ. This restricted shape is compatible with the expected shape *Expr*`[Γ/a.ΔO,b.ΔO/a.ΔL,b.ΔL/a.ΔI,b.ΔI][Boolean]` due to the standard weakening rule and the fact that the names bound by

a and b do not overlap (the iteration variables must be distinct, and the auxiliary variables must be fresh). We call such ECRs, whose expressions do not rely on Γ except in the outer bindings, "protected ECRs". One solution to the problem of section 5.1 is to protect the ECRs before merging; it is safe to merge protected ECRs.

The downside of this solution is that it introduces additional outer bindings. Normally these are not a problem, but when the merged ECRs correspond to the *inner* part of some other `in-nested` form (for a sequence with multiple levels of nesting), the added outer bindings become extra loop variables, which adds overhead that is difficult to optimize away.

In the next section we discuss an alternative solution that uses Racket's compile-time API.

4.5 The Problem with Nesting

There is another scoping error in our proposed implementation of `in-nested`. Consider the following example:

```
(define-syntax x (syntax-rules ())) ; macro that always raises error
(for ([z (in-nested ([x (in-list '((1 2 3) (4 5)))])
           x)])
  (printf "~a " z))
```

The name `x` is defined at the top level as a macro that always raises a syntax error. According to the scoping rules for `in-nested`, the reference to `x` as the `in-nested` expression's inner sequence expression should refer to the `x` bound in the preceding binding clause, which shadows the top-level `x` macro definition. That is, the inner `x` should refer to a list, which acts as a dynamic sequence, and this program should print the numbers from 1 to 5.

The actual behavior is a syntax error raised by the top-level `x` macro.

The problem is that we obtain the inner ECR by expanding the following binding clause:

```
[(z) x] ; should be ForClause[Γ0/{x:(Listof Nat)}][{z:Nat}]
```

That is, the type environment of the binding clause claims that `x` is a list-valued variable, because it occurs within the outer binding clause, and we will place the resulting ECR's components in the scope of a variable binding of `x`. But that is a *prediction*; at the time that we call `expand-for-clause`, the macro expander has not yet discovered and processed the new variable binding of `x`, and so `expand-for-clause` accesses the top-level macro definition instead. The same error would occur for any inner sequence expression whose expansion is strict in the binding of `x`—we expect such examples to be uncommon, but we should handle them correctly.

That is, `expand-for-clause` has the following pre-condition:

```
; expand-for-clause : Syntax[ForClause[Γ][Δ]] -> Syntax[ECR[Γ][Δ]]
; Pre-condition: Γ describes the macro expander's *current* environment
```

Note that the pre-condition depends on the type specification.

The solution is to use Racket's API for scopes and bindings (Flatt et al. 2012) to update the environment before expanding the inner binding clause. We create the following compile-time helper function:

```
; make-mark-as-variables : Syntax[(x:Id ...)]
;    -> (values Syntax[(y:Id ...)]
;               (Syntax[Expr[Γ/x:τ...][τ']] -> Syntax[Expr[Γ/y:τ...][τ']]))
; Post-condition: y... resolve as variables in the current environment
(define (make-mark-as-variables xs)
  (define defctx (syntax-local-make-definition-context))
```

```
(syntax-local-bind-syntaxes xs #f defctx)
(define (add-scope stx)
  (internal-definition-context-introduce defctx stx 'add))
(values (add-scope xs) add-scope))
```

The first line of the function body defines `defctx` as an environment rib. The second line registers the given identifiers in the rib as variables (the `#f` argument means the names are variables and not macros). More precisely, it generates a `y` identifier for each input `x` identifier and records that `y` is a variable. In the scope-set hygiene model (Flatt 2016), each `y` identifier is simply the corresponding `x` identifier with the rib's *scope* added to its *scope set*. The third line defines a function that marks the rib as in scope for the given expression. We return this function along with the new identifiers (`ys`).

The correct implementation of `in-nested` has the following steps: we collect the iteration variables of the outer binding clauses, and we use them as the argument to `make-mark-as-variables` to create a marking function. When we form the outer binding clauses and the inner binding clause, we apply the marking function to both

- the iteration variable lists (left-hand sides) of the outer binding clauses
- the inner sequence expression (right-hand side)

Note that the marking function is applied to both binding occurrences and expressions containing references, so the relationship between them is maintained. For example, for the program above, the outer binding clause is

```
[(x) (in-list '((1 2 3) (4 5)))]
```

and the inner binding clause is

```
[(z) x]
```

where the highlighted terms are processed by the marking function.

Once we have the outer and inner binding clauses, we expand them to ECRs, merge the outer ECRs as described before, and then nest the merged outer ECR with the inner ECR. The post-condition of `make-mark-as-variables` satisfies the pre-condition of `expand-for-clause`.

Using `make-mark-as-variables` has another consequence: it is unnecessary to protect the outer ECRs before merging them. The reason is that the identifiers created by `make-mark-as-variables` are effectively fresh, with one caveat discussed below. A scoping error in `merge` is possible when an iteration variable from one ECR is the same as a reference within another, so replacing the iteration variables with fresh names avoids the problem. The results of that process can be specified as follows:

$$\textit{FreshForClause}[\Gamma][\Delta][\Delta'] ::= \textit{ForClause}[\Gamma][\Delta] \quad \text{where } \mathrm{dom}(\Delta') \cap \mathrm{dom}(\Gamma) = \emptyset$$
$$\textit{FreshECR}[\Gamma][\Delta][\Delta'] ::= \textit{ECR}[\Gamma][\Delta] \quad \text{where } \mathrm{dom}(\Delta') \cap \mathrm{dom}(\Gamma) = \emptyset$$

We assert that `expand-for-clause` preserves this property:

```
; expand-for-clause : FreshForClause[Γ][Δ][Δ'] -> FreshECR[Γ][Δ][Δ']
; where Γ describes the *current* environment
```

And the naive `merge` operation is legal on such ECRs:

```
; merge : FreshECR[Γ][Δ][Δ'] ... -> ECR[Γ][Δ']
; where Δ' = Δ...
```

There is a caveat to considering the identifiers created by `make-mark-as-variables` as fresh. In fact, they differ from the input identifiers only by the addition of a single scope. The additional scope is normally sufficient to make them act as distinct names, but the scope-set identifier resolution algorithm has the following property: if `x` is an identifier and `y` is the same identifier with an

additional scope, then the term `(lambda (y) (lambda (x) y))` results in an "ambiguous binding" syntax error. We wish to avoid such errors, but we also want to avoid changing our model of names and type environments to expose the structure of identifiers and scope sets. So we add the following side-condition to the contract for `make-mark-as-variables`:

> If `x` is an identifier passed to `make-mark-as-variables` and `y` is the corresponding new identifier (`x` plus a scope), then we consider `y` as fresh, but we require that `x` must never occur in a type environment after `y`.

4.6 Generalization

Finally, we define a general sequence composition form called `do/sequence` that combines mapping, filtering, parallel iteration, and nested iteration. The syntax of `do/sequence` is like the syntax of a `for` loop: it takes a list of clauses (including both binding clauses and `#:when` clauses) followed by a body. For example, the `in-custom-set` form (Section 4) could be written as follows:

```
(do/sequence ([x (in-vector (custom-set-table s))]
              #:when (pair? x)
              [y (in-list x)])
  y)
```

The `do/sequence` can express both `fast-sequence-map` and `fast-sequence-filter`, represented by the `do/sequence` bodies and `#:when` clauses, respectively. For example:

```
(fast-sequence-map sqr
  (in-list '(1 2 3)))
=
(do/sequence ([x (in-list '(1 2 3))])
  (sqr x))

(fast-sequence-filter odd?
  (in-list '(1 2 3)))
=
(do/sequence ([x (in-list '(1 2 3))]
              #:when (odd? x))
  x)
```

Furthermore, whereas `fast-sequence-map` is limited to single-valued sequences, `do/sequence` support multi-valued sequences. For example:

```
(do/sequence ([(key val) (in-hash (hash 'a 1 'b 2 'c 3))])
  (values key (if (odd? val) 'odd 'even)))
```

The `do/sequence` form is decomposed into `in-nested`, a simpler variant of `in-when` from Section 5.1, and a generalized form of Racket's built-in `in-value` form. Some combinations can be optimized by handling them as a single unit. For example, nesting can be implemented with lower overhead when one of the sequences is either `in-when` or `in-value`.

In short, `do/sequence` combines the power of the four features—nested iteration, parallel iteration, mapping, and filtering—into a single tool for composing sequences, without sacrificing the laziness principle from Section 2.2 or the ability to handle multiple-valued elements.

5 PERFORMANCE

Our goal was to define fast sequence forms that would provide the expressiveness of dynamic sequences and the performance of fast sequences. Ideally, they should have close to the performance of the solution using a `for` loop without sequence abstractions. To check if the forms defined in previous sections achieve these goals, we've run microbenchmarks for measuring their performance. The measurements confirm that our fast sequence forms are significantly faster than dynamic sequences. Furthermore, they show good performance as compared to the `for` loop solution.

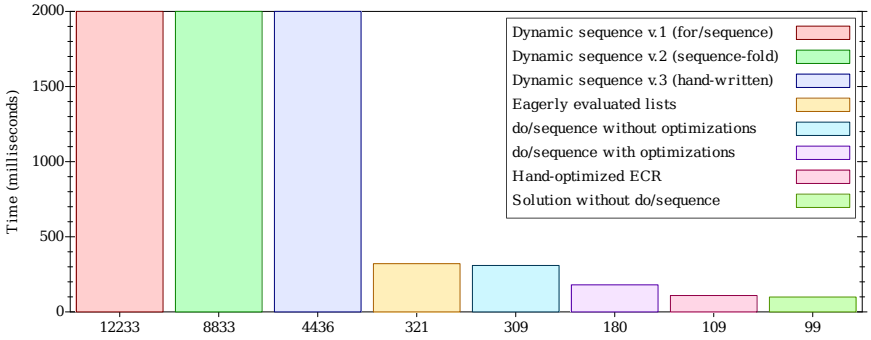

Fig. 5. Performance comparison (Racket 8.3 CS)

We've compared our fast sequence forms with different solutions using dynamic sequences and eagerly evaluated lists. Figure 5 shows results of benchmarking those kinds of sequences specifically for nesting. (The first three bars in the histogram are truncated in order to make differences between smaller bars more visible; the smallest truncated bar is over twice the height of the plot.) Two dynamic sequences in the benchmarks are the uses of `for/sequence` and `sequence-fold`, and one is hand-written using Racket's interface for defining dynamic sequences, where iteration is described by a pair of positions in the outer and inner sequences.

The eager list solution uses specialized list functions, so contains only one layer of dynamic dispatch at the end. We emphasize it is *eager* because it computes the entire list before starting the first loop iteration. The unoptimized version of `do/sequence` from Section 4 is barely faster than the list version, but significantly faster than the dynamic sequences. Our decomposition strategy for `do/sequence` adds overheads; we have introduced optimizations to eliminate some of the overhead by fusing common patterns (such as binding chunks followed by when chunks). The optimizations further improve the performance. We've also compared `do/sequence` with a solution implemented by a hand-optimized "perfect ECR" that further eliminates some nested `let` expressions introduced for fixing scoping. As the benchmark results in Figure 5 show, the optimized version of `do/sequence` is close to the "perfect ECR" solution, although we would like to explore additional optimizations to close the gap further. The last bar of the histogram shows the result for the solution without sequence abstraction—that is, using multiple `for` clauses and body. We believe that the limitations of the sequence interface may prevent any implementation of `do/sequence` from achieving the performance of the multi-clause solution.

6 RELATED WORK

Racket's `for` loops and sequences are based on SRFI-42's Eager Comprehensions (Egner 2005), although the surface syntaxes and APIs are different. Both `:parallel` in SRFI-42 and Racket's corresponding `in-parallel` have variants of the `merge` bug discussed in Section 5.1. SRFI-42 supports nesting within comprehensions, similar to Racket's nesting rules for chunks of `for` clauses, but it does not provide a `:nested` combinator for sequences (or "generators," in its terminology); see "Why is there no `:nested` generator?" in Egner (2003).

Shivers (2005) presents a macro-implemented and macro-extensible `loop` syntax inspired by the `loop` syntax of Common Lisp. A loop clause is translated to a sequence of "loop toolkit" (LTK) units, which contain "control flow graphs" (CFGs) that are linked and placed in a template similar to Racket's loop skeletons. The LTK units are similar to ECR components, but they are more flexible,

and they do not have a fixed binding structure. Consequently, in comparison to a Racket binding clause, a `loop` clause can have wider effects on the loop's control flow and result accumulation, but its binding structure is not predictable solely from its position in relation to other loop clauses. The author provides the example of the clause `(initial (i 0 (+ i 1)))`, which binds the loop variable `i` and also contains a reference to the same variable in the update expression. The author devises a scoping rule based on the underlying control-flow graphs, "binders dominate references," and presents a type-like scoping system for the CFG layer. The scoping system is not lifted to the LTK level or the loop clause level, though, and the paper does not connect the scoping rules for CFG languages to the Scheme macros and templates that express and manipulate CFGs. The `loop` macro supports nested sub-loops, but it does not appear to directly support specification of a flat iteration space expressed via nested iterations as our `in-nested` and `do/sequence` do.

Romeo (Stansifer and Wand 2014) is a language for binding-safe metaprogramming. The language is similar to Pottier (2007), with a first-order functional core extended with types that represent binding structures and `fresh` and `open` expressions, but its type system richer and it provides a stronger correctness guarantee (α-equivalence, rather than absence of unbound names). Romeo does not directly model the Scheme-style hygienic macro expansion process; rather, a macro expander with a fixed set of hygienic macros can be modeled as a Romeo program. Romeo's binding types are based on λ_m (Herman 2010). Adams (2015) similarly explored α-equivalence as a fundamental characteristic of hygienic macro expansion.

Romeo's type system corresponds closely with our binding specifications, but where we express scoping through type environments and meta-identifiers (syntax variables, not concrete variables), Romeo expresses scoping through products containing identifier sets and indexes. Romeo's import ($\downarrow$) annotation is like the type environment parameters of our `Expr` and `ECR` nonterminals, except that $\downarrow$ specifies only extensions relative to the ambient environment; Romeo's export ($\Uparrow$) annotation is like the Δ parameter. Romeo requires an SMT solver to handle disjointness (#) constraints, which implies that our system would be at least as difficult to automate.

Turnstile (Chang et al. 2020; Chang et al. 2017) is a DSL for implementing typed languages in Racket by using Racket's macro expander to implement type checking and type-based translation to core Racket expressions in a single recursive pass. We consider the relevance of Turnstile in two distinct senses. First, is Turnstile suitable for the implementation of our sequence macros, such as `do/sequence`? No, that would not achieve our primary goal. Turnstile is useful for implementing type systems, but it does not help prove their correctness. Furthermore, our macros exist in a context and manipulate sub-forms that are outside of Turnstile's influence, so it would be unable to check the actual uses of our macros. Second, would Turnstile be useful for implementing a typed meta-language for expressing our macros? Yes, it would, but we must finish the type system's design first.

7 CONCLUSION

We have implemented support for composing new sequence macros from existing sequence macros, thus improving their abstraction capabilities without losing their performance advantage over dynamic sequences.

To aid the implementation process, we have articulated a framework for specifying type and binding structure for DSL nonterminals, macros, and compile-time helper functions. We used an informal type discipline based on this framework to find our scoping mistakes and understand their solutions. In the process we gathered information about what sorts of reasoning we needed to accommodate. We hope to continue to refine this system into a generally useful tool for reasoning about macros.

BIBLIOGRAPHY

Michael D. Adams. Towards the Essence of Hygiene. In *Proc. Symposium on Principles of Programming Languages (POPL '15)*, pp. 457–469, 2015. https://doi.org/10.1145/2676726.2677013

Michael Ballantyne, Alexis King, and Matthias Felleisen. Macros for Domain-Specific Languages. *Proc. ACM on Programming Languages (OOPSLA '20)* 4, 2020. https://doi.org/10.1145/3428297

Stephen Chang, Michael Ballantyne, Milo Turner, and William Bowman. Dependent Type Systems as Macros. In *Proc. Symposium on Principles of Programming Languages (POPL '20)*, 2020. https://doi.org/10.1145/3371071

Stephen Chang, Alex Knauth, and Ben Greenman. Type Systems as Macros. In *Proc. Symposium on Principles of Programming Languages (POPL '17)*, pp. 694–705, 2017. https://doi.org/10.1145/3009837.3009886

Ryan Culpepper. Fortifying macros. *Journal of Functional Programming* 4-5(22), pp. 224–243, 2012.

Sebasian Egner. SRFI 42: Eager Comprehensions. Scheme Requests for Implementation, 2003. https://srfi.schemers.org/srfi-42/srfi-42.html

Sebastian Egner. Eager comprehensions in Scheme: The design of SRFI-42. In *Proc. Workshop on Scheme and Functional Programming (Scheme '05)*, pp. 13–26, 2005. See also https://srfi.schemers.org/srfi-42/.

Matthew Flatt. Binding As Sets of Scopes. In *Proc. Symposium on Principles of Programming Languages (POPL '16)*, pp. 705–717, 2016. https://doi.org/10.1145/2837614.2837620

Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that Work Together: Compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming* 22(2), pp. 181–216, 2012. https://doi.org/10.1017/S0956796812000093

David Herman. A Theory of Typed Hygienic Macros. PhD dissertation, Northeastern University, 2010.

Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic Macro Expansion. *Symposium on Lisp and Functional Programming*, pp. 151–161, 1986.

Francois Pottier. Static Name Control for FreshML. In *Proc. Symposium on Logic in Computer Science (LICS '07)*, pp. 356–365, 2007. https://doi.org/10.1109/LICS.2007.44

Olin Shivers. The Anatomy of a Loop: A Story of Scope and Control. In *Proc. International Conference on Functional Programming (ICFP '05)*, pp. 2–14, 2005. https://doi.org/10.1145/1086365.1086368

Paul Stansifer and Mitchell Wand. Romeo: A System for More Flexible Binding-Safe Programming. In *Proc. International Conference on Functional Programming (ICFP '14)*, pp. 53–65, 2014. https://doi.org/10.1145/2628136.2628162