Embedding Concurrent Generators

Peter Mills
Department of Computer Science
University of Idaho
phmills@acm.org

Clinton Jeffery
Department of Computer Science
University of Idaho
jeffery@uidaho.edu

***Abstract*—Generators are a natural fit for expressing concurrency. In dynamic languages such as Icon and Unicon where every expression is a generator that produces a sequence of values until it fails, there is a pervasive opportunity for exploiting concurrency. In this paper we present a simple model of explicit concurrency for generators based on co-expressions and multithreaded generator proxies, called pipes. Co-expressions are coroutines that shadow the environment to prevent interference, while pipes are generator proxies that communicate with the original expression running in a separate thread using blocking queues. Together these two mechanisms are sufficient to express both interleaving as well as true concurrency in the form of parallel pipelining, and to build higher-order abstractions such as map-reduce. We then present techniques for embedding concurrent generators into Java using transformation. We introduce a form of annotations for mixed-language embedding, called scoped annotations, that allow mixing in functionality at the level of expressions, methods, or classes. Transformations over the annotated regions then unravel the syntax of generator expressions to a conventional form by flattening nested generators in order to enable native evaluation and seamless interoperability. Mixed-language embedding allows using the succinct notation of concurrent generators within a familiar object-oriented setting, and enables their use for high-level coordination as well as the prototyping and refinement of parallel programs for multi-core architectures.**

Keywords—generators; Icon; Unicon; mixed-language; multi-paradigm; program transformation.

I. INTRODUCTION

Goal-directed evaluation is a computational paradigm that combines the power of generators with backtracking search. In goal-directed evaluation every expression is a generator that produces a sequence of values or fails, and operations search to find successful results over the product space of their operands. Introduced in the influential dynamic language Icon [1] and later refined in its object-oriented descendent Unicon [2], goal-directed evaluation and its pervasive use of generators are potentially a natural fit for expressing concurrency. However, while such a paradigm can succinctly express search, several challenges remain in its effective application in parallel computation.

The first challenge lies in developing concurrency abstractions that mesh with pervasive generators and are cleanly amenable to implementation. To answer this challenge, in this paper we present a minimalist set of concurrency mechanisms for Unicon that accommodates both Icon's coroutines, called co-expressions, as well as multithreaded communication between them using pipes. Co-expressions are coroutines that shadow the environment to prevent interference, while pipes are generator proxies that communicate with a co-expression running in a separate thread using the put and take operations of blocking queues. Together these mechanisms are sufficient to express parallel pipelining, and to build higher-order abstractions such as map-reduce.

While it may seem a bit of an oxymoron to introduce concurrency into a dynamic language characterized by relaxed typing and dynamic dispatch, the benefits lie in improving performance, in enabling the prototyping and exploration of parallel algorithms as well as their iterative refinement, and in the use of concurrent generators for high-level coordination among larger-grained processes expressed in other languages. In particular, the latter benefits strongly argue for the capability to embed goal-directed evaluation into other more efficient object-oriented languages that support parallelism. Such a capability would expand the reach of generators within a familiar setting and allow their use for coordination as well as refinement. In contrast to other dynamic languages that support parallelism [3,4,5,6], our goal is to support mixed-language programming across fundamentally differing computational paradigms rather than just to support multiple paradigms of concurrency within a single language. However, a capability for grafting goal-directed evaluation onto existing languages in a manner that provides seamless interoperability is a difficult challenge.

To answer the second challenge we present a novel approach to embedding goal-directed evaluation and its concurrency mechanisms into existing object-oriented languages based on program transformation. We introduce a form of annotations for mixed-language embedding, called scoped annotations, that allow mixing in Unicon functionality at the level of expressions, methods, or classes. Transformations over the annotated regions then unravel the syntax of generator expressions to a conventional form by flattening nested generators and making iteration explicit in order to enable native evaluation. The transformations are benign in that they are largely oblivious to the grammar of the surrounding language and leave code foreign to Unicon unchanged, while the mechanisms used to unravel the syntax to a conventional form provide seamless interoperability with other object-oriented languages.

In previous work [7] we demonstrated the utility of the approach for a sequential core of Unicon by implementing the transformations for Java as well as its dynamic analogue Groovy, and housed them in an interpretive harness called Junicon that realizes both an interactive extension of Groovy

Proc. 21st International Workshop on High-Level Parallel Programming Models and Supportive Environments,
30th IEEE International Parallel & Distributed Processing Symposium (IPDPS HIPS '16), 366-375.
DOI doi.ieeecomputersociety.org/10.1109/IPDPSW.2016.8

as well as a translator of embedded goal-directed evaluation into Java. In this paper we extend the transformational approach to yield an implementation of co-expressions, as well as a new technique for coordination based on the notion of multithreaded generator proxies that are layered over co-expressions using blocking channels. Our work presents a model of concurrent generators that simplifies and unifies the thread-based model previously developed for Unicon in [8], and enables its embedding into other object-oriented languages.

The contributions of this paper are as follows. First, we present a simple model of explicit concurrency for generators based on co-expressions and multithreaded generator proxies. We demonstrate the utility of the model in expressing parallel pipelining, as well as building higher-order abstractions such as map-reduce, in a mixed-language setting that uses scoped annotations to specify the embedding of concurrent generators. Second, we present techniques for transforming concurrent generators into Java, as well as its dynamic analogue Groovy, that leverage Java's facilities for multi-threaded concurrency. The techniques rely on flattening nested generators for primary expressions so as to enable grafting goal-directed evaluation onto other languages, as well as on synthesizing co-expressions and multithreaded generator proxies. Mixed-language embedding allows using the succinct notation of concurrent generators within a familiar object-oriented setting, and enables their use both for the high-level coordination of processes in other languages, as well as the prototyping and refinement of parallel programs for multi-core architectures. In particular, the capability for interactive evaluation under Groovy further enables exploration and rapid prototyping.

The remainder of this paper is organized as follows. Section 2 first provides more detailed background on Icon and Unicon. In Section 3 we present a model of concurrency for generators. In Section 4 we describe scoped annotations and illustrate their application to expressing parallel pipelining and map-reduce within a mixed-language setting. In Section 5 we describe the transformations that flatten generator expressions and translate concurrent generators into Java. Section 6 provides details on the implementation, while Section 7 provides the results of benchmarking concurrent generators when translated to Java. Lastly, Section 8 reviews related work, and we present our conclusions in Section 9.

II. BACKGROUND

At the heart of Icon and Unicon is the notion of a generator, which is an expression whose evaluation lazily yields a sequence of values, i.e., generates them one at a time on demand. The notion of generator functions has its origin in the language CLU [9], where a function can yield a result and suspend until the next value is needed. In Icon and Unicon this concept is extended into a dynamically typed notation that combines the pervasive use of generators with backtracking search.

A. Goal-Directed Evaluation

In Icon every expression is a generator that produces a sequence of values or fails, and nested generators are implicitly composed by mapping functions or operations over the cross-product of their arguments, and then filtering to find successful results. For example, consider the simple expression that finds multiples of prime numbers in a given range:

```
(1 to 2) * isprime (4 to 7)
```

where *isprime*(*x*) is defined to produce *x* if it is prime, and otherwise fail, and the *to* construct produces a range of numbers.

The above expression will, for each value in the sequence (1,2), iterate through each value in the second sequence (4,5,6,7) and for the latter that are prime numbers, yield their product. The compound expression itself forms a generator that, at each iteration, searches to find the next successful result, and so produces the sequence 1*5, followed by 1*7, then 2*5, then 2*7. Such search has particular application in string processing, the forte of Icon and Unicon.

The implicit composition of nested generators in Icon may be more clearly understood by decomposing it in terms of Icon's product operator,

```
e & e'
```

which for each *i* in *e*, iterates over each *j* in *e'*, and yields *j* as the next successful result of iteration. In other words,

```
e & e' ≡ filter(succeed (for i in e { for j in e' { j } }))
```

Function application, *f*(*e*,*e'*), is then equivalent to

```
f(e,e') → (i in e) & (j in e') & (k in f(i,j))
```

where (*i* in *e*) denotes bound iteration that assigns each value in the iterator sequence for *e* to a variable *i*, and where *f* is a generator function that produces a sequence of values on invocation. Applying the above transformation to operators, (*i*=*e*) can further be seen to be equivalent to (*i* in *e*).

The above example of prime multiplication can thus be recast as an iterator product:

```
i=(1 to 2) & j=(4 to 7) & isprime(j) & i*j
```

which corresponds to the Python generator expression:

```
(i*j for i in range(1,2) for j in range(4,7) if isprime(j))
```

and represents nested iteration. Conversely, a Python generator expression

```
f(x) for x in S if P(x)
```

is equivalent to

```
(x=S) & P(x) & f(x)
```

Generator expressions are also closely related to Java streams as well as monad comprehension [7]. In the terminology of Java streams,

```
f(e) → e.flatMap(x->f(x)).filter(succeed)
```

The above examples highlight how goal-directed evaluation combines generators with the concept of success and failure. An expression, at each iteration, succeeds and produces a value, or fails and terminates the iterator, which in turn fails. In other words, generators, when viewed as Java iterators, are terminated by failure of the *next*() method. Moreover, at each iteration, an operation will generally be performed only if the operands all succeed, and otherwise it

fails. Thus, expression evaluation is conditioned on the success of its terms. For example, *f*(*x*,*y*) will fail if either of the arguments *x* or *y* fails, and so the call to *f* will not occur. Similarly, in the iterator product *x* & *y*, if at a given iteration point the precondition *x* fails, then *y* is not evaluated. The & operator is thus fundamental as it embodies notions both of cross-product as well as conditional evaluation.

It is important to note that, since every expression is a generator, their composition, and the program in toto, just yields one large iterator. Even the familiar sequence construct, *a;b;c*, denotes the concatenation of iterators that runs through *a* and *b* as singleton iterators that are limited to producing at most one result, called bounded expressions, and then delegates remaining iteration to the last term *c*. Actual iteration, i.e., executing the iterator's *next*(), only occurs at the outermost level of interaction, for example in class field initializers, and in the main method of a program.

Icon and Unicon further provide a first-class reference semantics in which expressions can yield variable names to be assigned. Similarly function names used in invocation can themselves be generator expressions. For example,

```
(f | g)(x)
```

where | means concatenation of generators, is equivalent to:

```
f(x) | g(x)
```

and so iterates first through *f*(*x*) and then *g*(*x*). The above implies that method references, or some form of lambda abstraction, may be required for implementation.

Goal-directed evaluation thus embodies a conventional syntax with a very unconventional meaning. The provision of implicit search as well as the reference semantics, while powerful, pose challenges in embedding within other languages.

B. Co-expressions and Threads

Icon and Unicon also provide support for interleaving as well as true concurrency in the form of coroutines and multithreaded interaction, respectively. A coroutine is an expression that can suspend and transfer control to another expression, and when called again will resume at the point of suspension with its environment intact. Icon incorporates a notion of coroutines, called co-expressions, that further preclude interference by copying local variable references upon creation, and that are explicitly stepped on each iteration using an @ operator. Previous work on concurrency in Unicon [8] extends this notion with co-expressions that can run in a separate thread, and communicate with each other through a variety of synchronization mechanisms including explicitly created blocking channels and mutexes. While sufficient to express a wide variety of parallel interaction, communication must be made explicit, and to date there is no mechanism to naturally chain together generators at a high level.

III. Model of Concurrency

Generators are a natural fit for expressing concurrency. In dynamic languages such as Icon and Unicon where every expression is a generator, there is a pervasive opportunity for exploiting concurrency. While their composition under goal-directed evaluation offers the potential for chaining generators together in parallel, no model has been proposed which leverages this potential. In this section we present a simple model of explicit concurrency for generators that naturally transforms the composition of generators into parallel pipelines tied together using blocking queues. The spartan set of concurrency operators are sufficient to express true concurrency in the form of parallel pipelining, as well as to build higher-order abstractions such as map-reduce.

<> e	First-class generator.
|<> e	Co-expression that shadows the local environment.
|> e	Generator proxy that runs in a separate thread.
@ c	Next, i.e., step co-expression one iteration.
! c	Promote co-expression to a generator.
^ c	Restart with a new copy of the local environment.

where *e* is an expression, and *c* is a co-expression.

Figure 1. Calculus for concurrent generators.

Figure 1 presents the minimalist set of operators for concurrent generators. In Figure 1, the model of concurrency is based on a single unified notion of first-class iterators that must be explicitly stepped to the next iteration. The simplest first-class operator lifts a given expression into a singleton iterator that returns the original generator, in other words,

```
<>e → new Iterator() { next() { return e; } }
```

Explicitly stepping the first-class generator is through the @ operator:

```
@e → e.next()
```

Promoting the first-class entity back to a generator is through the ! operator:

```
!e → new Iterator() { next() { return @e; } }
```

which simply unravels it, or equivalently

```
!e → repeatUntilFailure(suspend @e)
```

since the composed iterators must be suspendable. Lastly, the restart operator ^ resets the iterator to its beginning state.

A. Co-expressions.

A co-expression is similar to a first-class iterator, but in addition creates a copy of its local environment, i.e., it shadows any referenced method local variables and parameters. In other words,

```
|<> e → ^(<>e)
```

where the refresh operator ^ for co-expressions is refined as follows, using Java's notation for lambda expressions and where (*x*,*y*,*z*) are locals:

```
^e → ((x,y,z)-> <>e) (((()->[x,y,z])())
```

Co-expressions thus minimize interference by isolating a copy of the local environment. In addition, as with a normal coroutine, the @ activation operator will transfer control between co-expressions so as to interleave the threads of execution.

B. *Generator Proxies.*

Lastly, a pipe is simply a generator proxy for a co-expression that runs in a separate thread and iterates until failure, and that uses a blocking channel for the communication of results. A blocking channel, or blocking queue, has put and take operations that wait until the queue of results is not full or not empty, respectively. Each iteration of the proxy will wait until a result has been placed in the channel by the co-expression running in the separate thread. Thus the surrounding expression runs in parallel to the piped expression. In other words,

```
|>e → new Iterator() { next() { new Thread { run() {
        c=|<>e; while (!fail) { out.put(@c); }}}.start() }}
```

where *out* is the output blocking queue, and an @ operation on a pipe is *out.take*(). The output blocking queue is a plain Java *BlockingQueue*, and is exposed as a public field to permit further manipulation. Bounding the output queue buffer size can also be used to throttle a threaded co-expression.

In its simplest form, a singleton piped iterator that produces one result forms a future or mutable variable, whose put and take operations wait until the channel is empty or full respectively. Historically it has been well established that mutable variables as found in Id Noveau's M-structures [10], the M-Vars of Parallel Haskell [11] and Concurrent Haskell [12], and Linda's tuplespace operations [13], as well as in an earlier form in the single-assignment synchronization variables of CML [14] that wait on read until being defined, are a fundamental building block that can be used to build higher-order concurrency abstractions. Not surprisingly, the derived blocking queues of Java are similarly powerful building blocks, and while they do not preclude the beneficial use of other synchronization mechanisms, they do provide the basis of a minimalist framework for coordination.

The above calculus for concurrent generators is thus sufficient to express a wide variety of parallel computations. For example the simple expression

```
x  *  ! |> factorial(! |> sqrt(y))
```

will, for given generated sequences *x* and *y*, spawn off their factorial and square-root computations in parallel, effecting explicit task parallelism in the form of a pipeline. A pipeline consists of a chain of tasks where the output of each element is the input of the next, synchronized using some form of blocking queues. The above is in contrast to

```
x * y
```

which reflects implicit data parallelism over the two generator sequences, and is the type of aggregate operation naturally amenable to map-reduce. The term map-reduce [15] refers to a constrained parallel functional style whose aggregate operations consist of stages of map functors followed by an optional shuffle and then reduction. The paradigm typifies Java parallel streams, and is typically implemented by partitioning the stream and then having multiple worker threads perform data-parallel operations sequentially on the decomposed chunks of data.

For example, for a given chunk c, mapping a function *f* over the chunk would be expressed using generators as:

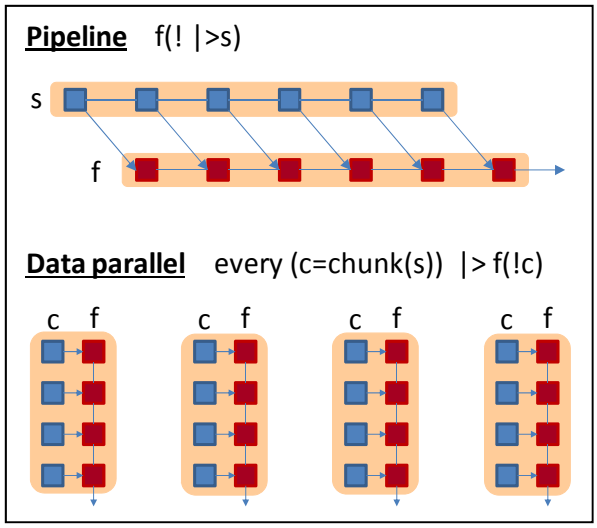

Figure 2. Pipeline and data-parallel models.

```
! |> f(!c)
```

where the ! operator lifts lists as well as co-expressions to iterators. The above formulation is subtly different from conventional map-reduce in that it enforces ordering between the results of the partitioned threads. However, such a formulation still requires the streams to be effectively splittable, i.e., have non-interference within a given stream operation.

The data-parallel decomposition of map-reduce thus differs from the calculus of concurrent generators: the former can be viewed as fixed-data that applies all pipeline stages to data distributed over threads, while the latter can be viewed as fixed-code that assigns a pipeline stage to each thread and exchanges data between them, using the terminology of [16]. Figure 2 illustrates the relationship between pipelining and data-parallel decomposition when specified using concurrent generators. In Figure 2, the tan oblongs represent separate threads of execution, which for pipelines encapsulate an entire stream, while for data-parallelism encapsulate a chunk of the source stream over which the function is mapped.

In the next section we examine how, in conjunction with multi-language integration, concurrent generators can be used to build and explore higher-order concurrency abstractions such as map-reduce.

IV. Multi-Language Integration

A primary goal of our research is to enable the use of concurrent generators, and goal-directed evaluation in general, within the broader scope of other languages that support parallel programming. In particular our approach to support multiple languages and paradigms is to specify embedding in a manner where the embedded regions are oblivious to the grammar of the surrounding context. In our implementation we do not need parsers for Java or Groovy. Rather we only need a general metaparser that recognizes complete statements, based on grouping delimiters such as braces and parentheses, in order to recognize embedded regions. Within a transformational framework, each

embedded region is then transformed and injected into the surrounding context, from the innermost outwards, to yield the final program. The exact transforms are dependent on both the embedded and surrounding language types.

To specify embedded regions we introduce a form of annotations, called scoped annotations, that blend Java annotations and XML. For example, scoped annotations of the form:

@<script lang="junicon"> x = f(g(y)); @</script>

are used to specify the embedded language, and delimit the sections of code where flattening of generator expressions occurs. Scoped annotations in general have the following admissible forms:

@<tag $attr_1$=x_1 ... $attr_n$=x_n> expression @</tag>
@<tag $attr_1$=x_1 ... $attr_n$=x_n/>
@<tag($attr_1$=x_1, ... ,$attr_n$=x_n)> expression @</tag>
@<tag($attr_1$=x_1, ... ,$attr_n$=x_n)/>

where the tag name may be qualified with either an XML namespace or Java package name, respectively. Like XML, such annotations can surround multiple statements, and can also be nested. Unlike conventional Java annotations that attach metadata to declarations or type use, scoped annotations can in addition modify expressions as well as arbitrarily delimited sections of code.

The above syntax of annotations for multi-language embedding is carefully chosen so that, similar to XML, it is attributed and scoped. The syntax is also chosen to be familiar and closely allied to Java annotations as well as other notations for scripted embedding such as HTML. However, the syntax is constrained so that its embedded use does not conflict with most programming language notations, nor does it collide with other forms of annotations such as for Java. In particular it differs from other notations such as JSP (Java Server Pages) in that, because it is tag-based, it can support multiple languages, and a single syntax supports both embedding as well as translator directives.

In a dual manner, a scoped annotation of the form @<script lang="java"> specifies native Java evaluation. When used outside a Unicon region, the latter exempts the section of code from being transformed, and so it is directly compiled, or if interactive is passed through to a Groovy script engine. When used within a Unicon region, it lifts the code into a singleton iterator over its closure, so it can participate in goal-directed evaluation.

A more detailed example that illustrates embedding concurrent generators into Java is given in Figure 3, showing a chain of mixed paradigm invocations. The program in Figure 3 takes lines of text, and computes a hash of the lines by splitting each line into words, converting the words into numbers, taking their square root, and then summing the result. Near the bottom of Figure 3, the Java method *runPipeline* can be seen to iterate over an embedded generator expression that spins off a pipeline to translate the words into numbers before computing their square root. The embedded expression returns a generator, exposed as a Java *Iterator* used in the *for* statement. The Unicon expression in turn cuts down to Java methods *wordToNumber* and *hashNumber*, as well as to a Unicon method *splitWords*.

```
class WordCount {
  static String[] lines;
  @<script lang="junicon">
    def readLines () { suspend ! lines; }
    def splitWords (line) { suspend ! ((String) line)::split("\\s+"); }
    def hashWords (line) {
      suspend this::hashNumber(this::wordToNumber(
            ! splitWords(line)));
    }
    def sumHash (sofar, hash) { return sofar + hash; }
  @</script>
  public Object wordToNumber (Object word) throws
                  NumberFormatException {
     return new BigInteger((String) word, 36);
  }
  public Object hashNumber (Object word) {
     return new Double(Math.sqrt(((Number) word).doubleValue()));
  }
  public void runPipeline () {
     double total = 0;
     for (Object i :
       @<script lang="junicon">
          this::hashNumber( ! (|> this::wordToNumber(
               ! splitWords(readLines()))))
       @</script>
     ) { total = total + ((Double) i).doubleValue(); };
  }
  public void runMapReduce () {
     double total = 0;
     DataParallel dp = new DataParallel(1000);
     for (Object i : dp.mapReduce(hashWords, readLines,
                                          sumHash, 0) {
        total = total + ((Double) i).doubleValue();
     };
  }
}
```

Figure 3. Embedding concurrent generators into Java.

Of particular note is that Unicon methods allow the use of a *suspend* statement to create a generator function, a feature otherwise missing from Java as well as Groovy. However, a subtlety arises in the invocation of generator functions. While Unicon methods return an iterator in a manner similar to embedded generator expressions, and so can be freely used within Java, they are exposed on the surface as method references in order to allow the use of function names in expressions. As functional interfaces they must be invoked with an explicit method name such as *apply*, and so their invocation must be differentiated from native Java method invocation, achieved by using :: for the latter.

At the bottom of Figure 3 is a map-reduce version of the *runPipeline* method, called *runMapReduce*. Its *mapReduce* method in turn is defined in Figure 4, and implements a simple variant of map-reduce defined in Section 3. In Figure 4, the first method *chunk* breaks up a source stream into chunks, each chunk being a list of fixed size. The second method *mapReduce* then, for each chunk derived using the generator function *s*, spins off a task to map the given function *f* over its elements, and then reduces the result with

```
class DataParallel {
  public DataParallel (int size) { this.chunkSize = size; }
  int chunkSize = 1000;
  @<script lang="junicon">
  def chunk(e) {                    # Partition e into chunks
    chunk = [];
    while put(chunk,@e) do {
      if (*chunk >= chunkSize) then { suspend chunk; chunk=[]; }};
    if (*chunk > 0) then { return chunk; };
  }
  def mapReduce(f,s,r,i) {       # Map f over s and reduce with r
      var c, t, tasks = [];
      every (c = chunk(<>s)) do {
          t = |> { var x=i; every (x=r(x, f(!c) )); x };
          ((List) tasks)::add(t);
      };
      suspend ! (! tasks);
  }
  @</script>
}
```

Figure 4. Building map-reduce using concurrent generators.

the reduction function *r* and initial value *i*. Lastly, the *mapReduce* method returns a generator over the results of each chunk.

Figures 3 and 4 illustrate several key features of our approach. First, goal-directed evaluation can be embedded at the method or expression level, as well as the class level if desired. Second, the embedding can be arbitrarily nested across differing languages. Third, the technique for embedding, in conjunction with the transformations described in Section 5, provides seamless interoperability when intermixing Unicon and Java. Specifically, native types can be transparently passed to and from Unicon, including class instances and collections such as lists, with fields accessed and methods invoked from either side.

The programs in Figures 3 and 4 demonstrate how embedding concurrent generators, and in general a dynamic language for goal-directed evaluation, can be used to explore and prototype the comparative performance of parallel algorithms. In particular, since the implementation of the calculus for concurrent generators leverages the Java facilities for task management and communication, their seamless integration with Java permits refinement in an iterative development methodology. As mentioned in the introduction and observed in evaluation, when using an embedded dynamic language, and in particular one supporting goal-directed evaluation, there is a tradeoff of performance for succinctness. The role of embedded generators in a parallel setting is thus envisioned to be one of exploration and prototyping, as well as potentially one of coordinating more computationally intensive pieces encoded in languages such as Java. In the next section we briefly describe the transformations that enable goal-directed evaluation to be embedded into other languages with seamless interoperability.

V. TRANSFORMATION

A fundamental challenge in embedding goal-directed evaluation is that it is based on such a differing evaluation paradigm that interoperability with other languages can be severely problematic. In this section we describe a novel approach to embedding Unicon into Java, and with little modification into its dynamic analogue Groovy, based on transformation. The key problem to be solved is how to unravel the syntax to a conventional form in a manner that enables native function invocation and maintains seamless interoperability with the surrounding target language.

Program transformation is a broad term that refers to changing the form of a program into another one that is semantically equivalent, or, for example in some cases of refinement, more specific. While program transformation encompasses translation, which includes compilation and interpretation, as well as the formal refinement of specifications and rephrasing, our focus here is on what is sometimes called migration, that is, translation into another language at the same level of abstraction [17]. The transformations that take Unicon into Java and Groovy are formalized as term rewriting rules, and so yield an operational semantics [18].

A. Normalization of Primary Expressions

The first step in the transformation that embeds Unicon into a conventional object-oriented language such as Java is the flattening or normalization of generator expressions. A key goal is to preserve type declarations and their use in function invocations and field references, so as to enable native evaluation mechanisms as well as seamless interoperability. For example, we would want class definitions, variable declarations and simple method invocations such as *o.f(x,y)* to be left largely unchanged in migrating from Unicon to Java, and avoid reflection mechanisms or instrumentation that might hinder interfacing with Java. Following the above line of argument, more complicated expressions in Unicon that embody nested generator expressions must be reduced to the above simple form in a manner that makes iteration explicit.

To make iteration explicit, we introduce an operator for bound iteration, and decompose nested generators into products of such bound iterators. Consider the following example of a primary expression, which involves field reference and indexing in addition to function application, and where functions are allowed to be expressions that resolve to method references:

$$e(e_x,e_y).c[e_i]$$

This can be equivalently reformulated as:

$$(f \text{ in } [\![e]\!]_{\mathcal{N}}) \;\&\; (x \text{ in } [\![e_x]\!]_{\mathcal{N}}) \;\&\; (y \text{ in } [\![e_y]\!]_{\mathcal{N}}) \;\&\; (o \text{ in } !\, f(x,y)) \;\&\; (i \text{ in } [\![e_i]\!]_{\mathcal{N}}) \;\&\; (j \text{ in } !\, o.c[i])$$

where $\mathcal{N}$ denotes the recursive application of the above transformation for flattening generators. In the above rewriting, for each step in the primary from left to right, generator expressions have been moved outside into bound iterators, and the pieces of the primary chained together using these bindings. The ! operator denotes lifting, which reifies a term and promotes it to an iterator. Lifting a variable

x turns it into a property with get and set methods, i.e., ()-> *x* and (*r*)-> *x*=*r*, and then wraps it in a singleton iterator, in order to enable it to be passed as an updatable reference. Lifting an invocation *f*(*x*) takes its closure and delegates iteration to the generator produced by its invocation. For plain Java methods, invocation just promotes the result to a singleton iterator.

The above reformulation, if applied recursively to a more complicated expression, extracts implicit generators and makes iteration explicit, reducing the expression to a normal form that is free of nested generators. The remaining residual expressions can then be evaluated using mechanisms native to the translation target.

B. Composing Suspendable Iterators

After normalization, the transformation of expressions proceeds by mapping constructs and operators onto a stream-like interface for composing suspendable iterators using functional forms such as product, concatenation, map, and reduce. Suspendable iteration refers to iteration in which, in addition to *next*, there is a *suspend* operation. In a tree of composed iterators, *suspend* will return a value that is propagated up as the result of the root iterator's *next*. The following iteration of the root will then resume at the point of suspension. In the absence of composition, *suspend* is equivalent to *next*.

A single Java class, *IconIterator*, implements the stream-like interface in a tightly knitted logic that provides iteration that is suspendable, failure-driven, and optionally reversible. While the *IconIterator* class implements the *java.util.Iterator* interface, it differs in that *hasNext*() tests for failure of *next*(), which terminates the iterator. After failure, the iterator is then restarted on the following *next*(). The kernel is optimized to statefully resume its point of suspension on a succeeding *next*(), incurring zero cost for suspends. Subtypes of the *IconIterator* class built using the stream operations are then used as abbreviations for constructs such as *while*.

C. Transformation of Classes

When embedding at the class level, a last stage in transformation maps class fields and methods into the Java class model, in a manner that supports interoperability while still accommodating Unicon's reference semantics. As mentioned previously, in Unicon, variables and subscripted collections can be passed as updatable references, and function names can be used in expressions. At the same time, Unicon class fields and methods need to be exposed in a manner that can be passed to and used by native Java methods, and conversely easily access foreign class fields and methods from within Unicon.

Our approach to solve the above problem is to expose variables in both plain and reified form while maintaining consistency between them. This duality allows Java code to use the plain form, while embedded Unicon code can use the reified form. For example, consider the following field declaration in Junicon:

```
local x;
```

This is equivalently transformed to:

```
Object x;
IconVar x_r = new IconVar(()->x, (rhs)->x=rhs);
```

Methods are similarly defined in plain form, and then exposed as method references with the same name. Since methods in Unicon are variadic, i.e., they can take any number of arguments, they are effectively translated into variadic lambda expressions that return an iterator. For example, consider the following method definition in Junicon:

```
method M(x,y) { body }
```

This is equivalently transformed to:

```
Object M = (VariadicFunction) this::M;
Object M (Object... args) { ⟦ body ⟧_T }
```

where $\mathcal{T}$ denotes the translation of the method body. Further details of the transformations for expressions and classes are provided in [7].

D. Synthesis of Co-expressions and Generator Proxies

To support concurrent generators, the transformations also synthesize co-expressions as well as multi-threaded generator proxies. For co-expressions as well as their multithreaded proxies, the local environment is shadowed as described in Section 3, which requires textually scoping up for referenced locals and creating a lambda expression around the generator that isolates these locals. Code for co-expression as well as proxy creation is then generated, invoking the suspendable iterator runtime. In the latter area, a single core class, *IconCoExpression*, provides a unified model for handling first-class generators as well as co-expressions and multithreaded proxies, and provides support for activating co-expressions, i.e., switching between coroutines, as well as thread creation and communication using blocking queues.

Figure 5 shows the result of applying the above transformations to a simple method for spawning a data-parallel computation using concurrent generators. The method in Junicon before translation is as follows:

```
def spawnMap (f, chunk) {
   suspend ! (|> f(!chunk));
}
```

In Figure 5, in the method body, transformation has unraveled generator expressions into the composition of iterators using forms such as product, embodied in *IconProduct* as well as similarly named classes for operations and function invocation. As can be seen in Figure 5, spawning a thread for a co-expression is transformed into an *IconCoExpression* constructor over a closure for invoking *f* that first copies the referenced local environment, *chunk*. The *IconCoExpression* then handles the ancillary mechanics of creating the thread, activating the closure within it, and coordinating the communication of results using blocking queues. Thread creation and allocation leverage Java's facilities for thread pool management and support for multi-core execution.

After translation to Java, the function body itself is an iterator constructor, so that the function when invoked will return an iterator. For optimization the iterator body is cached in a stack upon method return, and then reused. Since

```
MethodBodyCache methodCache = new MethodBodyCache();
public Object spawnMap = (VariadicFunction) this::spawnMap;
public IIconIterator spawnMap (Object... args) {
 // Reuse method body
 IconIterator body = methodCache.getFree("spawnMap_m");
 if (body != null) { return body.reset().unpackArgs(args); };
 // Reified parameters
 IconVar f_r = new IconVar().local();
 IconVar chunk_r = new IconVar().local();
 // Temporaries
 IconTmp x_1_r = new IconTmp();
 IconTmp x_0_r = new IconTmp();
 // Unpack parameters
 VariadicFunction unpack = (Object... params) -> {
  if (params ==  null) { params = IIconAtom.getEmptyArray(); };
  f_r.set((params.length > 0) ? params[0] : null);
  chunk_r.set((params.length > 1) ? params[1] : null);
  return null;
 };
 // Method body
 body = new IconSequence(new IconSuspend(
  new IconProduct(new IconIn(x_1_r, (
  new IconCoExpression( (Object... args_2) -> {
    // Reified parameters
    IconVar chunk_s_r = new IconVar().local();
    IconVar f_s_r = new IconVar().local();
    // Unpack parameters
    VariadicFunction unpack_4 = (Object... params) -> {
      if (params ==  null) { params = IIconAtom.getEmptyArray(); };
      chunk_s_r.set((params.length > 0) ? params[0] : null);
      f_s_r.set((params.length > 1) ? params[1] : null);
      return null;
    };
    // Method body
    IconIterator body_3 = new IconProduct(new IconIn(x_0_r,
      new IconPromote(chunk_s_r)), new IconInvokeIterator(()->
      ((VariadicFunction) f_s_r.deref()).apply(x_0_r.deref())));
    // Return body after unpacking arguments
    body_3.setUnpackClosure(unpack_4).unpackArgs(args_2);
    return body_3;
  },  () -> { return IconList.createArray(chunk_r.deref(),
  f_r.deref()); }).createPipe())), new IconPromote(x_1_r))),
  new IconNullIterator(), new IconFail());
 // Return body after unpacking arguments
 body.setCache(methodCache, "spawnMap_m");
 body.setUnpackClosure(unpack).unpackArgs(args);
 return body;
}
```

Figure 5. Transformation of concurrent generators to Java.

Unicon methods are variadic, the signature of the exposed method, shown at the top of Figure 5, is a variadic lambda expression that returns an iterator.

VI. Implementation

We have implemented the transformations for embedding Unicon into Java as well as its dynamic analogue Groovy, and housed them within a generic interpretive harness. A key aspect of providing support for multiple languages lies in the structure of the harness itself. The harness provides a cascading set of interpreters that at each stage transforms its input and either executes it on a script engine, such as for Groovy, or chooses another interpreter to pass to for further transformation. In particular the outermost instantiation of the harness is a meta-interpreter that detects the embedded language and its context using scoped annotations, and dispatches statements to the appropriate sub-interpreter for transformation.

The Junicon interpreter is an instantiation of the above harness implemented in Java, customized with a preprocessor, a Javacc LL(k) parser for Unicon that emits XML, transforms for normalization and translation to either Java or Groovy, and a Java kernel that implements the stream-like interface for composing suspendable iterators. By enabling embedding within either Groovy or Java, the interpreter can function both interactively and as a tool that can emit its output for compilation that is free of dependencies on Groovy.

VII. Evaluation

To evaluate the utility of the techniques for embedding concurrent generators, several variants of the program described in Figure 3 were compiled to Java, and their performance measured against equivalent Java stream-based programs. The suite of embedded Unicon programs consisted of a sequential word-count, a pipeline-parallel word-count that split the hash function into two tasks, a map-reduce word-count that spread the hash function and its summation reduction over chunks of data, and a data-parallel word-count that only differed in performing summation over the sequence returned from flattening the chunks, thus splitting out the reduction and effecting serialization. The suite of Java programs similarly consisted of a sequential word-count, a pipelined version built using *BlockingQueues* over two threads, a parallel stream-based version that implemented map-reduce, and a data-parallel version that was also stream-based but that split out the reduction. Both suites used arbitrary precision arithmetic, which is implicit in Unicon but must be made explicit in Java.

The Java Microbenchmarking Harness (JMH) was used to measure the performance of both suites on a Titan Quad AMD Opteron 6272 with 64-cores and 32GB of memory running Linux Fedora 20, with 20 warmup iterations and 20 test iterations. In addition, a second heavyweight set of variants of the programs in both suites was also benchmarked, which increased the complexity of the hash function components and so the weight of the threaded tasks, in order to explore the relative overhead of coordination using concurrent generators.

Figure 6 shows the relative performance of embedded concurrent generators when translated to Java. Execution time is normalized with respect to that of the Java parallel stream benchmark for each of the lightweight and heavyweight sets, respectively. Confidence intervals at 99%, shown by whiskers at the top of the histogram bars, showed negligible variance. In Figure 6, the eight histograms on the left use the lightweight versions of the *wordToNumber* and *hashNumber* functions described in Figure 3 that constitute the parallel computation nodes. On the right of Figure 6 are shown eight corresponding histograms that use the far more

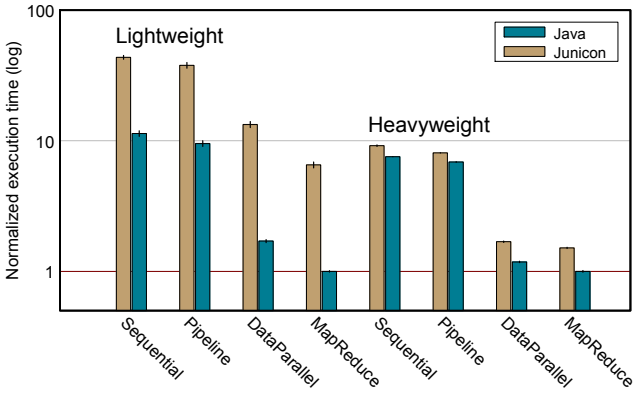

Figure 6. Performance when translated to Java.

heavyweight and computationally intensive hash functions, by a factor of roughly 80, achieved using trigonometry and prime number functions of Java's *Math* and *BigInteger* libraries.

The results showed that, as would be expected of a dynamic language, embedded generators yield worse performance than their native Java counterparts; however, the penalty is well under an order of magnitude. Moreover, as can be observed in the right of Figure 6, as the weight of the computational nodes increases, the relative overhead of the embedded concurrent generators significantly decreases. Indeed, even with map-reduce expressed entirely using concurrent generators, the performance impact on the right of Figure 6 is negligible. When used to coordinate complex tasks, concurrent generators may thus potentially provide performance roughly comparable to that of Java streams.

Another salient point is that the relative improvement among the embedded programs is roughly consistent with that of the comparable Java programs. For the purposes of exploration in a prototyping scenario, ideally it should be the case that the relative observed performance among experimental alternatives is preserved under refinement. While the benchmark results are preliminary and a proof of concept, they demonstrate the potential feasibility of exploration using concurrent generators.

VIII. Related Work

There are a variety of dynamic languages that support parallelism. These include Swift [3], which supports implicit parallelism in which every data-element is single-assignment and behaves like a future. Oz [4] is a multi-paradigm language for distributed programming whose explicit threads also use single-assignment dataflow variables. Julia [5] provides explicit task spawning and synchronization based on futures as well as blocking channels, and also supports metaprogramming through hygenic macros. Parallel Ruby [6] is similarly based on explicit task parallelism using futures as well as pipelines. In contrast to the above efforts, our approach is one of mixed-language embedding rather than multi-paradigm integration within a single language, and focuses on grafting a simple model of concurrent generators onto other languages through transformations that enable interoperability. We can thus leverage and integrate with the broader concurrency mechanisms of the underlying language, rather than directly incorporating multiple paradigms of concurrency.

The implicitly aggregate nature of goal-directed evaluation bears striking similarity to Java streams. Indeed, many of the stream composition operators such as map, reduce, and limit are either implicit or present as Icon primitives. Generators differ, however, in the support for suspendable iteration, as well as the provision of forms such as repeat and product that depend on the ability to be restarted. It bears noting that, while succinct, Java parallel streams are bound to a specific functional map-reduce style and implementation, which may hinder algorithmic exploration and may not be suitable to expressing other paradigms of concurrency.

The extension of Java iterators to support suspendable iteration bears some similarity to other work on interruptible iterators in JMatch [19]. There the focus was on extending coroutine iterators to handle update operations on the underlying data structure. Interruptible iterators for ML were also examined by Filliatre [20] using purely functional persistent cursors that also allow backtracking. In contrast, our iterators integrate suspend with failure-driven composition in a tightly knitted logic. It is also feasible to alternatively use multithreading to create a coroutine-like implementation of suspend in generator functions, as is provided in several Groovy and Java extension classes. Our techniques for embedding goal-directed evaluation not only enable the use of suspend in languages otherwise missing such a capability, but implement it without multithreading.

The challenges of formalizing and implementing goal-directed evaluation have given rise to a variety of research efforts. Principal among these efforts for Icon were continuation-based cross-translators [21, 22, 23, 24], a monad semantics for a small subset with compilation by partial evaluation [25], and a Java implementation called Jcon [26, 27]. The Jcon implementation in particular faced difficulties in transparently interfacing with other Java programs due to its instrumentation of data types and expressions with suspend and resume advice as well as its reliance on direct bytecode generation. In contrast to these efforts our transformations rely on flattening to do the work of instrumentation or higher-order functions used in monads, and so enable interoperability. Our research also addresses a wider set of concerns including generator propagation in an object-oriented setting. Under our approach certain problematic features of Icon and Unicon such as concurrency [8] become simpler to implement. Lastly, and perhaps most importantly, our research focuses on the larger problem of mixed-language embedding of goal-directed evaluation into other object-oriented languages.

IX. Conclusions

In this paper we introduced a simple model of explicit concurrency for generators, and developed a technique for embedding such concurrent generators into other languages based on program transformation. We presented a novel form of annotations, called scoped annotations, that are used

in conjunction with the transformations to support mixed-language integration. The transformations use flattening to unravel the syntax of pervasive generators to a conventional form, which is key to enabling interoperability. In the area of concurrency the transformations synthesize co-expressions that shadow the local environment, as well as synthesize multi-threaded generator proxies, and so enable succinctly expressing parallel pipelining. The transformations are implemented in an interpretive harness that can target Java as well as its dynamic analogue Groovy, and so realize a tool that can function both interactively and as a translator for compilation. In particular the provision for interactive evaluation enhances the ability for exploration and prototyping of parallel programs for multi-core architectures. We demonstrated the utility of the approach in building higher-order abstractions such as map-reduce, and evaluated their performance against equivalent Java stream-based programs.

Currently the implementation supports the full set of goal-directed constructs and operators, including those for concurrency and co-expressions, as well as most of Icon's built-in functions. Future efforts will focus on further evaluating and refining concurrency abstractions for generators. Lastly, program monitoring and debugging within a transformational framework is an area to be further explored.

ACKNOWLEDGMENT

The authors would like to thank Rob Kleffner for help in implementing many of Icon's built-in functions.

REFERENCES

[1] Griswold, R. E., Hanson, D. R., and Korb, J. T. 1981. Generators in Icon. *ACM Transactions on Programming Language and Systems* 3, 2, ACM Press, New York, NY, 144-161.

[2] Jeffery, C. L. 2001. Goal-directed object-oriented programming in Unicon. In *Proc. 2001 ACM Symposium on Applied Computing*. ACM Press, New York, NY, 306-308.

[3] Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., and Foster, I.T. 2011. Swift: A Language for Distributed Parallel Scripting. *Journal Parallel Computing* 37, 9, Elsevier Science Publishers, 633-652.

[4] Smolka, G. 1995. The Oz Programming Model. In *Computer Science Today*. LNCS 1000, J. van Leeuwen (Ed.), Springer-Verlag, 324-343.

[5] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V.B. 2014. Julia: A Fresh Approach to Numerical Computing. Technical Report arXiv:1411.1607, Cornell University Library Archive, Computer Science. http://arxiv.org/pdf/1411.1607v4

[6] Lu, L., Ji, W., and Scott, M.L. 2014. Dynamic Enforcement of Determinism in a Parallel Scripting Language. In *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 519-529.

[7] Mills, P. and Jeffery, C. 2016. Embedding Goal-Directed Evaluation through Transformation. To appear in *Proc. 31st ACM Symposium on Applied Computing*, ACM Press.

[8] Al-Gharaibeh, J., Jeffery, C., and Oikonomou, K.N. 2012. An hybrid model for very high level threads. In *Proc. 2012 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM Press, New York, NY, 55-63.

[9] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1977. Abstraction mechanisms in CLU. *Communications of the ACM*, 20, 8, ACM Press, New York, NY, 564-576.

[10] Barth, P. S., Nikhil, R. S. and Arvind. 1991. M-structures: Extending a parallel, non-strict functional language with state. In *Proc. 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS volume 523, Springer-Verlag, 538-568.

[11] Nikhil, R.S., Arvind, Hicks, J., Aditya, S., Augustsson, L., Maessen, J., and Zhou, Y. January 1995. pH Language Reference Manual, Version 1.0. Technical Report Memo-369, MIT Computation Structures Group, http://csg.csail.mit.edu/pubs/memos/Memo-369/memo-369.pdf

[12] Peyton-Jones, S., Gordon, A., and F. Sigbjorn, 1996. Concurrent Haskell. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, New York, NY, 295-308.

[13] Carriero, N., and Gelernter, D. 1989. Linda in Context. *Communications of the ACM*, 32, 4, ACM Press, New York, NY, 444-458.

[14] Reppy, J.H. 1991. CML: A higher-order concurrent language. In *Proc. Conference on Programming Language Design and Implementation*, ACM Press, New York, NY, 293-305.

[15] Dean, J., and Ghemawat, S. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51, 1, ACM Press, New York, NY, 107-113.

[16] Bienia, C., and Li, K., 2010. Characteristics of Workloads Using the Pipeline Programming Model. In *Proc. International Conference on Computer Architecture, Workshop on Emerging Applications and Many-core Architecture*, Springer-Verlag, 161-171.

[17] Visser, E. 2005. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation, Special issue on Reduction Strategies in Rewriting and Programming* 40, 1, Elsevier, 831-873.

[18] Serbanuta, T. F., Rosu, G., and Meseguer, J. 2009. A Rewriting Logic Approach to Operational Semantics. *Information and Computation* 207, 2, Elsevier, 305-340.

[19] Liu, J., Kimball, A., and Myers, A. C. 2006. Interruptible iterators. In *Proc. 33rd Symposium on Principles of Programming*. ACM Press, New York, NY, 283-294.

[20] Filliatre J.-C. 2006. Backtracking iterators. In *Proc. 2006 Workshop on ML*. ACM Press, New York, NY, 55-62.

[21] O'Bagy, J. and Griswold, R. E. 1987. A recursive interpreter for the Icon programming language. In *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*. ACM Press, New York, NY, 138-149.

[22] Allison, L. 1990. Continuations implement generators and streams. *The Computer Journal* 33, 5, Oxford University Press, 460-465.

[23] Walker, K., and Griswold, R. 1992. An Optimizing Compiler for the Icon Programming Language. *Software Practice and Experience* 22, 8, Wiley, 637-657.

[24] O'Bagy, J., Walker, K., and Griswold, R.E. 1993. An operational semantics for Icon: Implementation of a procedural goal-directed language. *Computer Languages* 18, 4, Elsevier, 217-239.

[25] Danvy, O., Grobauer, B., and Rhiger, M. 2002. A unifying approach to goal-directed evaluation. *New Generation Computing* 20, 1, Springer-Verlag, 53-73.

[26] Proebsting, T. A. 1997. Simple translation of goal-directed evaluation. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 1-6.

[27] Proebsting, T. A. and Townsend, G. M. 2000. A new implementation of the Icon language. *Software: Practice and Experience* 30, 8, Wiley, 925-972.