A Transformational Interpreter for Goal-Directed Evaluation

Peter Mills and Clinton Jeffery

Unicon Technical Report 17
June 12, 2014

Abstract

We develop a Java-based interpreter for the Unicon programming language using transformation, first into an iterator calculus and from there into the dynamic language Groovy. In Unicon every expression is a generator that produces values until it fails, and operations are conditioned on success and failure. The transformations first normalize primary expressions by flattening nested generators and making iteration explicit. Control constructs are then translated into an iterator calculus for composing suspendable generators. Lastly, methods are mapped onto the Java class model using variadic lambda expressions. The transformations, expressed in XSLT, are also retargeted to Java to enable later compilation.

Unicon Project
http://unicon.org

Department of Computer Science
University of Idaho
Moscow, ID, 83844, USA

A Transformational Interpreter for Goal-Directed Evaluation

PETER MILLS and CLINTON JEFFERY, University of Idaho

We develop a Java-based implementation of the Unicon programming language using high-level program transformation, first into an iterator calculus and from there into the dynamic language Groovy. Unicon is an object-oriented descendent of Icon, a unique language where every expression is implicitly a generator that iteratively produces a value until it fails, and where operations are conditioned on the success of their operands. To align Unicon with native invocation mechanisms, the transformations first reduce primary expressions to a normal form that flattens nested generators and makes iteration explicit. Control constructs and operations are then translated into an iterator calculus that composes suspendable generators using forms such as product, concatenation, map, reduce, and exists. The calculus is implemented as a compact Java kernel that presents a stream-like interface. Lastly, Unicon methods as well as Icon procedures are mapped onto the Java class model using variadic lambda expressions. The transformations are implemented using XSLT (XML Language for StyleSheet Transformations), a rule-based language for transforming XML documents, and housed in a generic transformational interpreter. The interpreter, which we call Junicon, functions both interactively and, with only slight modification to the transformations, as a tool that translates its input directly to Java for later compilation that is free of dependencies. Such a transformational approach realizes a lightweight and retargetable implementation that can seamlessly integrate with and leverage the full range of Java capabilities, including its portability and facilities for concurrency and graphics.

Categories and Subject Descriptors: **D.3.1 [Programming Languages]**: Formal Definitions and Theory; **D.3.2 [Programming Languages]**: Language Classifications – Very high-level languages, Icon; **D.3.4 [Programming Languages]**: Processors – Code generation, Interpreters, Retargetable compilers; **I.1.3 [Symbolic and Algebraic Manipulation]**: Languages and Systems – Evaluation strategies; **I.2.2 [Symbolic and Algebraic Manipulation]**: Automatic Programming – Program transformation

General Terms: Languages, Design, Theory

Additional Key Words and Phrases: Unicon, generators, iterator calculus, program migration, XSLT, Groovy, Java

1. INTRODUCTION

The goal-directed evaluation paradigm underlying Icon [Griswold et al. 1981; Griswold and Griswold 1996] and its object-oriented descendent Unicon [Jeffery 2001; Jeffery 2013a; Jeffery 2013b] poses formidable challenges in implementation. Icon is a unique language where every expression is implicitly a generator, and where evaluation of an expression is conditioned on the success or failure of its components. For example, the meaning of the simple expression "f(1 to 2)" is an iterator that yields the results of f(1), followed by f(2), and then failure since (1 to 2) has terminated and failed. The notion of generator functions has its origin in the

Author's addresses: P. Mills (corresponding author) and C. Jeffery, Department of Computer Science, University of Idaho, Moscow, ID.

language CLU, where a function can yield a result and then suspend until the next value is needed [Liskov et al. 1977; Liskov et al. 1981]. In Icon this concept is extended into an extremely compact and dynamically typed notation that implicitly composes nested generator expressions, and where such iterators are terminated by failure.

As such the semantics of Icon, and in turn its implementation, is fairly complex, as evidenced in various techniques that have been investigated for its translation. One notable effort in particular focused on a Java bytecode generator for Icon called Jcon [Proebsting and Townsend 2000] that employed a Prolog-like Byrd-box model using fail and resume ports [Proebsting 1997]. A number of other studies have looked at continuation-based approaches for implementation [O'Bagy and Griswold 1987; Allison 1990; O'Bagy et al. 1993] as well as for formally defining the semantics of Icon, including a denotational semantics [Gudeman 1992] and a semantics based on list and continuation monads [Danvy et al. 2002]. Despite these efforts, to some degree the semantics remains clouded and not obvious to the implementer, and this complexity is reflected in the difficulty of achieving malleable implementations. The Jcon implementation in particular heavily instrumented data types and expressions with suspend and resume advice and relied on direct bytecode generation, an approach which did not permit transparently interfacing with other Java programs or libraries, or tracking evolving Java technology. Since Unicon is implemented as a cross-translator into Icon that treats classes as records, similar problems of seamless integration with Java arise using the above implementation techniques.

We investigate an alternative approach for implementing Unicon based on program transformation, first into an iterator calculus that distills the essential concepts in goal-directed evaluation, and from there into another high-level Java-based scripting language, Groovy. The iterator calculus captures a minimal set of forms for composing suspendable and failure-driven iterators including product, concatenation, map, reduce, and exists, and so combines aspects of both functional languages and predicate logic. This calculus forms the basis for rewriting rules that first reduce primary expressions such as function invocation and object field reference to a normal form that is free of nested generators and makes iteration explicit in order to enable native evaluation. The rewriting rules then equationally translate control constructs and operations into that simpler basis. A compact kernel in Java implements the iterator calculus, and serves as the final target of the transformation rules. To realize the transformations themselves, we employ a novel technique that uses XSLT (the XML Language for StyleSheet Transformations), a rule-based language for transforming XML based on XPath pattern matching, to rewrite XML abstract syntax trees and then deconstruct them into a Groovy program.

Using the above approach of XSLT-based program transformation, we have implemented a Java-based interpreter for Unicon, called Junicon. Such a transformational interpreter realizes several advantages. By reducing generator expressions to a recognizable and explicit form, we clarify the semantics of Unicon so that it can be understood in terms of conventional programming language concepts, and enable the grafting of goal-directed behavior onto other languages. By transforming onto another high-level Java-based dynamic language in a careful manner that preserves types such as lists using their Java equivalent, we can seamlessly integrate with and leverage the full range of Java capabilities. Lastly, the expressiveness of XSLT for program transformation enhances the ability to retarget the implementation. The ease of retargeting is demonstrated by parameterizing the transforms to emit Java rather than Groovy code. Using these two sets of transforms, the interpreter is able to act either interactively, or as a tool that can translate its input to Java for later compilation that is free of dependencies.

In the remainder of this paper we first provide more detailed background on Icon and Unicon, as well as Groovy. We then describe the iterator calculus and the transformations from Unicon into it, both for reducing primary expressions to a normal form free of implicit generators, and for translating control constructs and classes. We examine how, with only slight modification, the transforms can be concretely retargeted to Java rather than Groovy. We then illustrate how the rewriting rules are expressed in XSLT. We also describe the design of the generic transformational interpreter underlying our implementation that supports multi-stage transformations and pluggable execution substrates. The results of benchmarking Junicon when translating to Groovy as well as Java are described, and show performance comparable to that of native Unicon. Lastly we review related work, and conclude with a summary of key contributions.

2. BACKGROUND

Icon is a unique programming language designed as the successor to the pattern matching language SNOBOL [Griswold et al. 1971]. At the heart of Icon 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. As in CLU and later languages such as C# [Jagger et al. 2007] and Python [Rossum and Drake 2011], generators in Icon can be constructed using generator functions that use a "suspend" statement – corresponding to CLU's "yield" – to return a value and on next invocation to resume at the point of suspension. For example, the following Icon procedure, which corresponds to the "to" construct mentioned above, on invocation will result in a generator that produces an ascending sequence of values:

```
procedure range (from, bound)
    local count;
    count := from;
    while (count <= bound) do { suspend count; count +:= 1 }
end
```

Thus, the expression "range(1,2)" yields the values 1 and 2, and then fails. Such generators can then be used in lieu of collections in loops and list comprehension. While Icon uses the term generator, we will use the term iterator interchangeably with it, and will distinguish it from a Java iterator when necessary.

However, Icon goes a step beyond conventional languages in its pervasive use of generators. In Icon *every* expression is a generator[1], and nested generators are *implicitly* composed by mapping functions or operations over the cross-product of their arguments. For example, the expression

```
f(range(1,2), range(3,4))
```

will, for each value in the first operand range(1,2), iterate through each value in the second operand range(3,4), and then iterate through each value in the result of applying *f*. If *f* yields a single value, the above results in the sequence: f(1,3), f(1,4), f(2,3), f(2,4). The implicit composition of nested generators in Icon may be more clearly understood by decomposing it in terms of Icon's product operator,

[1] Strictly speaking, in Icon parlance a generator is an expression that can produce multiple results. In this paper we use the term generator to include expressions that only produce at most one result.

```
e & e'
```

which for each *x* in *e*, iterates over each *y* in *e'*, and yields *y* as the result of iteration. The above example can thus be recast as an iterator product:

```
i := range(1,2) & j := range(3,4) & k := f(i,j)
```

which corresponds to the Python generator expression:

```
(k for i in range(1,2) for j in range(3,4) for k in f(i,j))
```

and represents nested iteration.

It further bears noting that in Icon procedures are first class citizens, and 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.

Icon then combines generators with the concept of success and failure to realize goal-directed evaluation. An expression, at each iteration, succeeds and produces a value, or fails and terminates the iterator, which in turn fails. In other words, iterators are terminated by failure of the *next*() method. Moreover, at each iteration, an operation will typically 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, the expression:

```
x := if (k > 0) then k
```

will perform an assignment to *x* only if *k* > 0, since otherwise the "if" expression fails, which in turn causes the assignment to fail. In Icon, the concept of true or false is thus replaced with success and failure, and failure propagation is analogous to a bottom-preserving or undefined-preserving semantics. Failure propagation similarly applies to function invocation as well as to operations such as product. For example, "f(x,y)" will fail if either of the arguments *x* or *y* fails, and thus not be invoked. Similarly, in the iterator product operation "*x* & *y*", if at a given iteration point the precondition *x* fails, then *y* is not evaluated. Thus the & operator 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 in control constructs, and in the program in toto, from a semantic viewpoint just yields one large iterator. Even the familiar sequence construct, *a*;*b*, is special in that it denotes the concatenation of iterators, with all but the last iterator forced to be a singleton limited to producing at most one result before failure. The net effect of the sequence construct is thus to run through each singleton iterator, called a bounded expression, until failure, and then delegate remaining iteration to the last term. Actual iteration over a composed iterator expression, i.e., executing the iterator's *next*(), only occurs at the outermost level of interaction. These points occur at interpreted statements outside a class definition, in class field initializers, and in the main method of a program.

Icon further provides an expressive reference semantics that supports lazy dereferencing. Variable references are treated as first-class citizens in generator expressions, and are only dereferenced when needed, for example as arguments to operators or methods, the latter which thus have call-by-value semantics. For example, in the expression:

```
(if i > j then i else j) := 0
```

the value 0 is assigned to *i* or *j* depending on their comparison. Under the hood, the generator on the left-hand-side of the assign produces a reference to *i* or *j*, which is then used by the assignment operator. Indexing operations are also first class citizens, in that an index such as *c*[*i*] is maintained as an offset into a collection until its value is needed, and so provides an updatable reference.

Icon also provides a notion of reversible assignment, *x* <- *y*, which on more than one iteration, reverses the assignment and then fails. For example, in the unbounded expression (*x* <- *y*) & *e*, if *e* fails, *x* will revert to its original value. Reversible assignment, along with string scanning, is one of the few vestiges of implicit data backtracking in Icon, since elsewhere state is not saved or restored. Icon's treatment of variable and index references as first-class citizens implies that some form of reification is needed when transforming to another high-level language.

Unicon in turn is an object-oriented extension of Icon which provides support for classes with multiple inheritance in a manner similar to that of C++. Unicon is currently implemented by a preprocessor that translates its programs into Icon by treating classes as records, and so any challenges in implementing a Java version of Icon hold for it as well.

In contrast, Groovy [Dearle 2010] is a fairly conventional object-oriented dynamic language that extends the pre-lambda version of Java [Gosling et al. 2014] with parameterized closures, and which is implemented by compilation into Java bytecode that runs on any Java virtual machine. Like Icon, Groovy is dynamically typed, in other words variables need not be declared as having a type. Groovy also provides seamless integration with Java, in that Groovy class instances and data types can be transparently passed to and from Java, with class fields accessed, and similarly methods invoked, from either side. A notable feature of Groovy is its provision for parameterized closures, which expresses lambda abstraction, also called lambda expressions in Java. For example,

```
def  f = {x,y -> x+y}
```

defines *f* as a function formed from the closure of the expression to the right of the arrow, and with parameters *x* and *y*. A closure in Groovy always returns the last argument. As expected, f(1,2) will thus yield the value 3. The above features, in particular its relaxed typing and its provision of closures, make Groovy an attractive translation target for Unicon.

While Unicon is a powerful language whose dynamic typing makes it easy to use, there is to date no viable Java implementation of it, nor arguably a clear operational semantics of its underlying goal-directed evaluation that would drive such an implementation. Yet a Java implementation of Unicon, or more precisely an interpreter for Unicon that would run within the Java Runtime Environment, has many potential advantages. These advantages include portability, access to Java concurrency and graphics utilities, the use in web applications, and the potential to integrate into the increasingly widespread Java-and-Linux-based Android Operating System for mobile handhelds. In contrast to earlier efforts for a Java based Icon implementation, our research focuses on higher-level cross-translation that maintains consistency with the Java type system.

3. TRANSFORMATION OF UNICON INTO GROOVY

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 [Feather 1987; Reddy 1990; Li 2010]. 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 [Visser 2005]. It bears emphasizing that the techniques described in this paper are made possible by having either the transformation source or target be a dynamically typed language; in the presence of static typing and complex type systems, the problem of transformation is vastly more complex.

The transformation of Unicon into Groovy is broken down into four stages: a transformation $\mathcal{N}$ for normalization of primary expressions, a transformation $\mathcal{T}$ that translates larger expressions including control constructs and operations into an iterator calculus, a transformation $\mathcal{K}$ that concretizes the iterator calculus into Groovy, and finally a transformation $\mathcal{C}$ that handles classes and methods.

3.1 Normalization of primary expressions

A key goal in the transformation of Unicon into Groovy is to maximally preserve type declarations and their use in function invocations and field references, so as to enable the use of native evaluation mechanisms and their concomitant optimization, as well as seamless integration with Java. Here, field reference means reference to an object's fields using dot notation. For example, we would want the class definitions, variable declarations and simple method invocations such as "o.f(x,y)" to be left largely unchanged in migrating from Unicon to Groovy, and avoid elaborate reflection mechanisms or extensive instrumentation that might preclude optimizations or 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:

    f(g(x))

This can be equivalently decomposed into:

    (i in x) & (j in g(i)) & (k in f(j))

where & denotes iterator product, and (*i* in *e*) denotes bound iteration that assigns each value in the iterator sequence for *e* to a variable *i*. The final result of the above expression will be a sequence whose values are bound to *k*. It bears noting that, in general, the iterator product and bound iteration operators used in the above decomposition are sufficient to express lazy list comprehension. For example, a Python generator expression

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

is equivalent to

    (x in S) & P(x) & f(x).

A similar flattening technique can be applied to more complicated expressions involving field reference and indexing in addition to function application, and where functions are allowed to be expressions that resolve to method references. Consider the following example of a primary expression:

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

This can be equivalently reformulated as:

(f in e) & (x in e_x) & (y in e_y) & (o in f(x,y)) & (i in e_i) & (j in o.c[i])

In the above rewriting, for each step in the primary from left to right, generator expressions have been moved outside into explicit bound iterators, and the pieces of the primary chained together using these bindings. The final result of the above expression will be a sequence whose values are bound to *j*. 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, avoiding more complicated and potentially costly mechanisms such as reflection or extensive instrumentation. In particular, by leaving index operations in their native form, the above approach potentially allows leveraging advanced capabilities such as those found in Groovy that blur the distinction between objects and maps, and so for example allow field access using o["f"]. Normalization thus aligns Unicon with a more conventional semantics for list comprehension and method invocation, clarifying its meaning as well as placing it into a form more amenable to native evaluation.

In general the above rewriting is applied to arbitrarily complex primary expressions such as:

e.f(x,y).c[i](z)

that consist of a combination of field reference, invocation, and indexing, and whose terms are identifier and literal atoms as well as generator expressions. Primary expressions also include collection literals such as [*x*,*y*] and [*k*:*v*,...] for list and map construction, respectively. Since expressions may evaluate to method references, one may also chain method invocations and indexing together, e.g., *f*(*x*)[*i*](*y*). It bears noting that previous semantic treatments [Gudeman 1992; Danvy et al. 2002] did not explicitly address function application using such method expressions, nor did they address propagating generators through the fields in object references. Thus the above formulation of normalization is a step forward in making clear the semantics of generator propagation.

The syntax for the subset of Unicon that is to be normalized is shown in Figure 1. We slightly extend Unicon syntax to incorporate several useful features such as local declarations within blocks, lambda expressions, allocation using *new C*(*e*) in addition to Unicon's function-like construction using *C*(), and method references using *o*::*m*. As is further discussed in Section 4, the *new* construct as well as method references are provided to support accessing native Java classes and methods from within Junicon. Primary expressions, which consist of field references, function invocation, and indexing as well as identifiers, literals, and collections, are shown in the bottom of Figure 1.

The rewriting rules that reduce primary expressions to normal form are shown in Figure 2, and define the normalization transform $\mathcal{N}$. We make the normalization transform $\mathcal{N}$ independent of the later transforms for control constructs and classes, so as to enable staging them separately. Although it is feasible to equivalently define

E ::= Control | Block | Closure | Operation | Primary
Control ::= if E_1 then E_2 [else E_3]
| E_1 to E_2 [by E_3] | every E_1 [do E_2]
| while E_1 [do E_2] | until E_1 [do E_2]
| repeat E | not E
| suspend E_1 [do E_2] | return [E] | fail
Block ::= { E_1 ; ...; E_n } | { local y_1[:= E_y^1]; …; local y_m[:= E_y^m]; E_1 ; ...; E_n }
Closure ::= { (x_1,…,x_p) -> local y_1[:= E_y^1]; …; local y_m[:= E_y^m]; E_1 ; ...; E_n }
Operation ::= E_1 & E_2
| E_1 | E_2
| E_1 op E_2
| op E where op in +, -, :=, …
| new Dotname(E_1, …, E_n)
Primary ::= identifier | literal
| (E) | [E_1, ..., E_n]
| [E_k^1:E_v^1, ..., E_k^n:E_v^n]
| E(E_1, ..., E_n)
| E[E_1, ..., E_n]
| E.identifier
| Dotname::identifier
Dotname ::= identifier | Dotname.identifier

where x_i, y_i are identifiers

Fig. 1. Syntax of expressions to be normalized.

$\mathcal{N}$ to be recursively dependent on $\mathcal{T}$, the above independence of $\mathcal{N}$ is desirable, since normalization reflects aligning generator expressions with a more conventional semantics, and $\mathcal{N}$ can then be used as a standalone transformation to graft such a capability onto other languages. The latter is the approach taken in our implementation, and is reflected both in the XSLT transforms as well as the staged structure of the generic transformational interpreter, described later.

The normalization transform thus begins by, for a given program consisting of larger expressions such as control constructs and operators, descending into primary expressions through a default rule that recurses down through non-primaries otherwise leaving them unchanged. For a non-primary expression or construct c composed of terms t_i, the rule is a homomorphism over its terms:

$$[\![c\langle t_1,\ldots,t_n\rangle]\!]^p_{\mathcal{N}} \rightarrow c\langle [\![t_1]\!]^p_{\mathcal{N}},\ldots,[\![t_n]\!]^p_{\mathcal{N}}\rangle$$

We define $\mathcal{T}$ to analogously leave primaries alone, since they will have already been normalized. The normalization transform also has preprocessing rules that change certain constructs such as "new" and "to" into function calls, so that their arguments can be normalized. In particular the "to" construct is a prototypical generator function, since unlike other constructs it returns a generator when given non-generator arguments. Such ersatz function invocations are later rewritten back into allocation and range expressions respectively.

For a given primary expression, $\mathcal{N}$ then proceeds left to right along the fields and arguments inside it, decomposing field references and invocations into separate iterator product steps, and extracting complex fields and arguments into bound iterators. Along the way, $\mathcal{N}$ carries the accumulated object reference p, or prefix, to be used as the function or collection name in the decomposed invocation and indexing steps.

In the rewriting rules, lifting is denoted by $!\,x$, which reifies x and promotes it to an iterator, while $<x>$ denotes dereference of a reified value. Lifting a variable x

$\llbracket \mathrm{e} \rrbracket_{\mathcal{N}} \rightarrow \llbracket \mathrm{e} \rrbracket_{\mathcal{N}}^{\emptyset}$ where e is any expression **// Entry into transforms using empty prefix**

$\llbracket \mathrm{c}\langle \mathrm{t}_1,\ldots,\mathrm{t}_n \rangle \rrbracket_{\mathcal{N}}^{p} \rightarrow \mathrm{c}\langle \llbracket \mathrm{t}_1 \rrbracket_{\mathcal{N}}^{p}, \ldots, \llbracket \mathrm{t}_n \rrbracket_{\mathcal{N}}^{p} \rangle$ where c is a non-primary with terms t_i **// Descend into primaries**

$\llbracket \mathrm{e}_x \text{ to } \mathrm{e}_y \text{ by } \mathrm{e}_z \rrbracket_{\mathcal{N}}^{p} \rightarrow \llbracket \mathrm{range}(\mathrm{e}_x, \mathrm{e}_y, \mathrm{e}_z) \rrbracket_{\mathcal{N}}^{p}$ **// Synthetic functions for higher-order generators**

$\llbracket \text{new C(e)} \rrbracket_{\mathcal{N}}^{p} \rightarrow \llbracket \text{C.new(e)} \rrbracket_{\mathcal{N}}^{p}$

$\llbracket \mathrm{e.e'} \rrbracket_{\mathcal{N}}^{p} \rightarrow (\mathrm{o} \text{ in } \llbracket \mathrm{e} \rrbracket_{\mathcal{N}}^{p}) \;\&\; \llbracket \mathrm{e'} \rrbracket_{\mathcal{N}}^{p'}$ where p'= <o>, the dereference of o // **Field reference**

$\llbracket \mathrm{e(e')e''} \rrbracket_{\mathcal{N}}^{p} \rightarrow (\mathrm{o} \text{ in } \llbracket \mathrm{e} \rrbracket_{\mathcal{N}}^{p}) \;\&\; \llbracket \mathrm{<o>(e')e''} \rrbracket_{\mathcal{F}}$ where $\mathrm{e''=(e_1'')\ldots[e_i'']\ldots}$ or $\mathrm{[e_1'']\ldots(e_i'')\ldots}$ // **Invoke**

$\llbracket \mathrm{p(e_1,\ldots,e_n)e''} \rrbracket_{\mathcal{F}} \rightarrow (\mathrm{x_1} \text{ in } \llbracket \mathrm{e_1} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; \ldots \;\&\; (\mathrm{x_n} \text{ in } \llbracket \mathrm{e_n} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; (\mathrm{o} \text{ in } !\, \llbracket \mathrm{p(<x_1>,\ldots,<x_n>)} \rrbracket_{\mathcal{O}}) \;\&\; \llbracket \mathrm{<o>e''} \rrbracket_{\mathcal{F}}$

$\llbracket \mathrm{p(e_1\ldots\,,\,,\,\ldots e_n\,)} \rrbracket_{\mathcal{O}} \rightarrow \mathrm{p(e_1\ldots\,,\,omit\,,\,\ldots e_n\,)}$ where skip last product if e''= Ø

$\llbracket \mathrm{e[e']e''} \rrbracket_{\mathcal{N}}^{p} \rightarrow (\mathrm{o} \text{ in } \llbracket \mathrm{e} \rrbracket_{\mathcal{N}}^{p}) \;\&\; \llbracket \mathrm{<o>[e']e''} \rrbracket_{\mathcal{F}}$ where $\mathrm{e''=(e_1'')\ldots[e_i'']\ldots}$ or $\mathrm{[e_1'']\ldots(e_i'')\ldots}$ // **Index**

$\llbracket \mathrm{p[e_1,\ldots,e_n]e''} \rrbracket_{\mathcal{F}} \rightarrow \llbracket \mathrm{p[e_1]\ldots\,[e_n]e''} \rrbracket_{\mathcal{F}}$

$\llbracket \mathrm{p[e']e''} \rrbracket_{\mathcal{F}} \rightarrow (\mathrm{x} \text{ in } \llbracket \mathrm{e'} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; (\mathrm{o} \text{ in } !\; \mathrm{p[<x>]}) \;\&\; \llbracket \mathrm{<o>e''} \rrbracket_{\mathcal{F}}$ where skip last product if e''= Ø

$\llbracket \mathrm{x} \rrbracket_{\mathcal{N}}^{p} \rightarrow \mathrm{!p.x}$ where x is an identifier in the last field of an object reference **// Simple atom**

$\llbracket \mathrm{x} \rrbracket_{\mathcal{N}}^{\emptyset} \rightarrow \mathrm{!x}$ where x is an identifier or literal outside of a complex primary

$\llbracket \mathrm{e::f} \rrbracket_{\mathcal{N}}^{p} \rightarrow \llbracket \mathrm{e} \rrbracket_{\mathcal{N}}^{p}\mathrm{::f}$ **// Method reference**

$\llbracket \mathrm{[e_1,\ldots,e_n]} \rrbracket_{\mathcal{N}}^{p} \rightarrow (\mathrm{x_1} \text{ in } \llbracket \mathrm{e_1} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; \ldots \;\&\; (\mathrm{x_n} \text{ in } [\mathrm{e_n}]\rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; \mathrm{![<x_1>,\ldots,<x_n>]}$ **// List**

$\llbracket \mathrm{[e_k{}^1:e_v{}^1,\ldots,e_k{}^n:e_v{}^n]} \rrbracket_{\mathcal{N}}^{p} \rightarrow (\mathrm{k_1} \text{ in } \llbracket \mathrm{e_k{}^1} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; (\mathrm{v_1} \text{ in } \llbracket \mathrm{e_v{}^1} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; \ldots \;\&$ **// Map**
$(\mathrm{k_n} \text{ in } \llbracket \mathrm{e_k{}^n} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; (\mathrm{v_n} \text{ in } \llbracket \mathrm{e_v{}^n} \rrbracket_{\mathcal{N}}^{\emptyset}) \;\&\; !\llbracket \mathrm{<k_1>:<v_1>,\ldots,<k_n>:<v_n>}]$

where !*e* denotes lifting of a primary expression, which reifies *e* and promotes it to an iterator, and <*e*> denotes the dereference of a reified variable or value. Invocation delegates to the returned generator; otherwise lifting gives a singleton iterator that yields one result before failing.

We skip bound iterator creation (*o* in $\llbracket e \rrbracket_{\mathcal{N}}^{p}$) if *e* is a *simple term* consisting of an identifier or literal, or a field reference, method reference, or collection literal composed of only simple terms, and just use the prefixed original term *p*'= *p.e* (or *p*'=*e* if *p*=∅) instead of <*o*> in the above products.

We skip bound variable creation (*o* in $\llbracket e \rrbracket_{\mathcal{N}}^{p}$) if $\llbracket e \rrbracket_{\mathcal{N}}^{p}$ is a product that ends in an iterator with binding *b*, or is itself such an iterator, and just use $\llbracket e \rrbracket_{\mathcal{N}}^{p}$ with *p*'=<*b*> instead of <*o*> in the above products. For example, (*o* in (*b* in *e*)) becomes just (*b* in *e*).

We also skip bound variable creation (*o* in !*p*(<*x*>)) or (*o* in !*p*[<*x*>]) for the last invoke or index step in a primary, and just lift the last product term, since there is no further need for chaining. For example, x.f(y)[z] → (o in !x.f(y)) & !o[z].

Lastly, for efficiency we skip over redundant parenthesis, i.e., $\llbracket ((e)) \rrbracket_{\mathcal{N}}^{p} \rightarrow \llbracket (e) \rrbracket_{\mathcal{N}}^{p}$, and $\llbracket (e) \rrbracket_{\mathcal{N}}^{p} \rightarrow \llbracket e \rrbracket_{\mathcal{N}}^{p}$ if not inside a primary.

Fig. 2. Normalization of primary expressions.

concretely turns it into a property whose get and set methods are initialized using closures, e.g.,

```
get = {-> x}  ;  set = {rhs -> x=rhs }
```

and then wraps it in a singleton iterator that produces the property on iteration. Lifting *f*(*x*) takes the closure of *f*(*x*) and promotes it to a restartable iterator that delegates to the generator produced by its invocation, reifying the generator results as needed. For functions which are not generators, e.g., normal Java methods, the invocation is promoted to a singleton iterator. Lifting *c*[*i*] reifies the index operation and promotes its evaluation to a singleton iterator that returns an assignable reference. Lifting thus ensures, first, that arguments and results are treated uniformly as iterators. Second, its use of reification with closures accommodates

Unicon's reference semantics, enables generator expressions to be used as first-class citizens, and makes the iterators capable of being restarted on failure. Lifting can be seen as being analogous to the monad return operator.

The above transforms must be similarly applied to all invocation and indexing arguments when there are multiple arguments, for example in

$e(e_1,\ldots,e_n)$ or $e[e_1,\ldots,e_n]$

where multidimensional indexes are first reduced to chains of single index steps. Moreover, to optimize the number of bound iterators, we skip the creation of bound iterators (o in $[\![e]\!]^p_{\mathcal{N}}$) if e is a simple term consisting of an identifier or literal, or a field reference, method reference, or collection literal composed of only simple terms, and just use the prefixed original term *p.e* instead of <*o*> in the above products. We also skip the creation of bound iterators if $[\![e]\!]^p_{\mathcal{N}}$ ends in a created iterator, and use its last binding instead of <*o*>. These optimizations avoid synthesizing unnecessary bound iterators, and shorten the chain of iterator products. For example, for *e*(*x*,*e'*) where *x* is an identifier, the rewriting would yield:

```
(f in e) & (y in e') & (z in !f(x,y))
```

Since lifting only occurs when creating bound iterators, under the above optimizations only invoke and index, and simple terms such as identifiers, literals, and residual field references and method references that appear outside a primary field or argument, are lifted.

Normalization thus flattens nested generators into a more conventional form that expresses lazy list comprehension. In Icon and Unicon, since everything is a generator expression, normalization and generator propagation is pervasive, and so a statement or method is in effect one giant comprehension. However, we envision that it would also be useful to have the capability to limit where normalization and generator propagation occurs.

We enable such a capability through the provision of scoped annotations. Scoped annotations, also called X-annotations, are a novel syntax that blends Java annotations and XML, and have the following admissible forms:

```
@<tag attr₁=x₁ … attrₙ=xₙ> expression @</tag>
@<tag attr₁=x₁ … attrₙ=xₙ />
@<tag(x₁, … ,xₙ)> expression @</tag>
@<tag(x₁, … ,xₙ)/>
```

Unlike conventional Java annotations that modify type declaration or use, scoped annotations can in addition modify expressions as well as arbitrarily delimited sections of code. For example,

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

forgoes transformation and pipes the code to Groovy for native evaluation. The unique syntax of scoped annotations is driven in part by the fact that Unicon already uses the @ operator for its co-expression construct. Junicon uses such annotations as directives to guide interpretation, as well as to attach metadata to types and expressions.

In a dual manner, scoped annotations could also be used to selectively graft goal-directed evaluation onto other languages such as Groovy and Java. For example,

```
@<generator> x = f(g(y)); @</generator>
```

I ::= I_1 & I_2	Product, i.e., for each *i* in I_1 { for each *j* in I_2 }
| I_1 | I_2	Concatenation, i.e., {for each *i* in I_1}; {for each *j* in I_2}
| I_g -> I_1 | I_2	Choice, i.e. if exists(I_g) then iterate over I_1, else iterate over I_2
| I*	Repeat iterator as long as it produces a non-empty sequence, i.e., succeeds at least once
| I:n	Limit iterator to at most n results, then force failure
| ! P	Lift normalized primary expression, i.e., promote it to an iterator
| *x* in I	Bound iteration, i.e., bind variable x to iterator results
| reduce *op* I	At each iteration, iteratively combine results from I until it fails, and return a singleton result, or fail if the operator fails or the operand sequence is empty
| map *op* I	Map operator over iterator, or if product, over pairs of its operands map *op* (I_1 & I_2) is equivalent to (*x* in I_1) & (*y* in I_2) & (! <*x*> *op* <*y*>)
| forall I	Reduce by, at each iteration, iterating over I until failure, then fails (equivalent to *reduce noop I*)
| exists I	Succeed if non-empty, i.e., produces at least one successful result
| not I	At each iteration, succeed on failure, and fail on success
| suspend I	Suspend after each iteration, yielding the value from I, i.e., forces ancestors to revisit the argument's next() until it fails
| return I	Return exists(I), i.e., forces ancestors to succeed and then terminate
| fail	Constant iterator that always fails
| (I)	Parenthesized expression
| Closure()	Invoke closure, used to translate blocks into an iterator bound to local declarations

P ::= Atom | Atom(A_1,...,A_n) | Atom[A_1,...,A_n] | Closure **// Normalized primary**
Closure ::= { (x_1,…,x_p) -> local y_1;…; local y_m; I }
Atom ::= Name | Name.Dotname | Dotname::identifier **// Simple normalized primary**
Name ::= identifier | <identifier> | literal **// Simple identifier or literal**
| [A_1,...,A_n] | [A^1_k:A^1_v, ..., A^n_k:A^n_v]
Dotname ::= identifier | Dotname.identifier

where x_i, y_i are identifiers, A_i are atoms, and <*x*> = *x*.deref() is the dereference of a reified variable.

Fig. 3. Syntax of the iterator calculus.

could be used to delimit the sections of code where implicit generator propagation occurs, in effect providing a scope for forming a sequence comprehension.

3.2 The iterator calculus

The iterator calculus distills the essential concepts of goal-directed evaluation, and provides a minimal spanning set of operators for composing iterators into which Unicon expressions and control constructs can be translated. As can be seen from the preceding discussion, product, bound iteration, and lifting are the first entries in the calculus needed for normalization, and effect lazily evaluated list comprehension. The full calculus for iterator composition is shown in Figure 3. It bears noting that *map* and *forall* are included as convenience mechanisms for optimizing the implementation, and can be equivalently expressed using product and reduce, respectively.

A Java kernel implements the above calculus in a single compact class, IconIterator, that provides the core logic for iteration that is failure-driven, suspendable, restartable, and optionally reversible. While the IconIterator class implements the java.util.Iterator interface, it differs in that failure on *next*() terminates the iterator, as indicated by both an *isFailed* property and an enumerated return value of *fail*. After failure, the iterator is then restarted on the following *next*(). Unlike other language extensions that implement suspend in iterators using multithreading, such as can be found for Groovy and Java, in Junicon suspend is tightly integrated into the kernel. Suspend could quite simply be implemented by, on a *next*(), skipping down the iterator expression tree to the point of suspension, which only requires traversing left instead of right if the left child is suspended. However, we further optimize the kernel for an outermost expression to statefully resume to its point of suspension, thus incurring zero cost for suspends. The reduction of Unicon to an iterator calculus thus reflects a purely iterator-oriented view of its semantics, rather than one that is based on notions of continuation-based control backtracking.

Nested instantiations of subtypes of the IconIterator class, reflecting the iterator expression tree, then serve as the final target of the transformation rules. The transformation that concretely takes the iterator calculus into the Java kernel is denoted by $\mathcal{K}$. For example, $I \& J$ is translated as follows:

$[\![\text{I \& J}]\!]_{\mathcal{K}} \rightarrow$ new IconProduct($[\![\text{I}]\!]_{\mathcal{K}}$, $[\![\text{J}]\!]_{\mathcal{K}}$)

Interestingly, for operations such as $I+J$, instead of normalizing the expression into an iterator product

(i in I) & (j in J) & (k in ! <i>+<j>)

which would work, we instead build a map operation into the class performing the iterator product, IconIterator, so that it performs the operation at each product pair if the operands succeed. In monad terminology, the above corresponds to bind, which consists of join over map. Thus, the operation $I+J$ is translated to:

$[\![\text{I + J}]\!]_{\mathcal{T}} \rightarrow$ new IconProduct($[\![\text{I}]\!]_{\mathcal{K}}$, $[\![\text{J}]\!]_{\mathcal{K}}$).map(new IconOperator({x,y -> x+y}))

although for efficiency we actually only define operators once.

The iterator calculus is loosely derived from a combination of functional forms [Backus 1978], guarded commands such as in GCL [Dijkstra 1975] and CSP [Hoare 1978], and first-order predicate calculus. Unlike typical mechanisms for lazily evaluated sequence comprehension, the iterator calculus allows specifying comprehensions, i.e., the intensional properties defining a sequence, using more powerful first-order formulae in a manner similar to Z schemata [Spivey 1992] and SETL [Schwartz et al. 1986].

3.3 Translation of control constructs into the iterator calculus

The iterator calculus provides the basis into which control constructs and operations are translated. Most of the operations in the iterator calculus, with the notable exception of lift, reduce, and map, are also primitives in Unicon. Other Unicon constructs are straightforwardly transformed into compositions of constructors and methods that embody the kernel for the calculus.

The rules for the transformation $\mathcal{T}$ that translates program expressions into the iterator calculus are shown in Figure 4. As can be seen in the figure, the rules provide an equational definition of the semantics of Unicon control constructs. Since the behavior of these control constructs can be initially difficult to understand by the user, even if informally described in detail, a succinct and precise formulation of their meaning is advantageous.

For example, the "*every*" construct corresponds to *forall* or *reduce*, and iterates until failure. The sequence construct, where terms are separated by ";", is equivalently transformed into the concatenation of bound expressions, i.e., a singleton iterator with limit 1, with the result being an iterator over the last unbounded term. Operations are simply transformed into map over products of the operands. Lambda expressions, on the other hand, must move any initializers for local declarations into the function body before recursively transforming that part of

$[\![\{E_1; \ldots; E_n; E_z\}]\!]_{\mathcal{T}}$	$\rightarrow \text{forall}([\![E_1]\!]_{\mathcal{T}}{:}1 \mid \ldots \mid [\![E_n]\!]_{\mathcal{T}}{:}1) \mid [\![E_z]\!]_{\mathcal{T}}$	**// Sequence**
$[\![\{ \text{local } y_1{:=} E_y^1; \ldots; \text{local } y_m{:=} E_y^m;\ E_1;\ldots;E_n \}]\!]_{\mathcal{T}} \rightarrow$		**// Block**
	$\{ \text{->} \text{ local } y_1;\ldots; \text{local } y_m;\ [\![\{y_1 := E_y^1;\ldots; y_m{:=} E_y^m;\ E_1;\ldots;E_n \}]\!]_{\mathcal{T}} \}\ (\,)$	
$[\![\{ (x_1,\ldots,x_p) \text{ -> local } y_1{:=} E_y^1;\ldots; \text{local } y_m{:=} E_y^m;\ E_1;\ldots;E_n \}]\!]_{\mathcal{T}} \rightarrow$		**// Lambda expression**
	$!\ \{ (x_1,\ldots,x_p) \text{ -> local } y_1;\ldots; \text{local } y_m;\ [\![\{y_1 := E_y^1;\ldots; y_m{:=} E_y^m;\ E_1;\ldots;E_n;\ \text{fail}\}]\!]_{\mathcal{T}} \}$	
$[\![\text{every } E]\!]_{\mathcal{T}}$	$\rightarrow \text{forall}([\![E]\!]_{\mathcal{T}})$	**// Control constructs**
$[\![\text{every } E_x \text{ do } E_y]\!]_{\mathcal{T}}$	$\rightarrow \text{forall}([\![E_x]\!]_{\mathcal{T}}\ \&\ ([\![E_y]\!]_{\mathcal{T}}\ ;\text{fail}))$	
$[\![\text{while } E_x \text{ do } E_y]\!]_{\mathcal{T}}$	$\rightarrow \text{forall}(([\![E_x]\!]_{\mathcal{T}}{:}1 \text{ -> } ([\![E_y]\!]_{\mathcal{T}};\ \text{succeed}{:}1))^*)$	// where succeed = not(fail)
$[\![\text{until } E_x \text{ do } E_y]\!]_{\mathcal{T}}$	$\rightarrow [\![\text{while (not } E_x) \text{ do } E_y]\!]_{\mathcal{T}}$	
$[\![\text{repeat } E]\!]_{\mathcal{T}}$	$\rightarrow \text{forall}(([\![E]\!]_{\mathcal{T}};\ \text{succeed}{:}1)^*)$	
$[\![\text{not } E]\!]_{\mathcal{T}}$	$\rightarrow \text{not}(\text{exists}([\![E]\!]_{\mathcal{T}}))$	
$[\![\text{if } E_g \text{ then } E_x \text{ else } E_y]\!]_{\mathcal{T}}$	$\rightarrow [\![E_g]\!]_{\mathcal{T}} \text{ -> } [\![E_x]\!]_{\mathcal{T}} \mid [\![E_y]\!]_{\mathcal{T}}$	
$[\![\text{if } E_g \text{ then } E_x]\!]_{\mathcal{T}}$	$\rightarrow [\![E_g]\!]_{\mathcal{T}} \text{ -> } [\![E_x]\!]_{\mathcal{T}}$	
$[\![\text{return } E]\!]_{\mathcal{T}}$	$\rightarrow \text{return}(\text{exists}([\![E]\!]_{\mathcal{T}}))$	
$[\![\text{suspend } E]\!]_{\mathcal{T}}$	$\rightarrow \text{suspend}([\![E]\!]_{\mathcal{T}})$	
$[\![\text{suspend } E_x \text{ do } E_y]\!]_{\mathcal{T}}$	$\rightarrow (x \text{ in } [\![E_x]\!]_{\mathcal{T}})\ \&\ (\text{suspend}(!x)\ );\ [\![E_y]\!]_{\mathcal{T}}\ ;\text{fail})$	
$[\![\ \text{fail}\ ]\!]_{\mathcal{T}}$	$\rightarrow \text{fail}$	
$[\![E_x\ \&\ E_y]\!]_{\mathcal{T}}$	$\rightarrow [\![E_x]\!]_{\mathcal{T}}\ \&\ [\![E_y]\!]_{\mathcal{T}}$	**// Iterator calculus operators**
$[\![E_x \mid E_y]\!]_{\mathcal{T}}$	$\rightarrow [\![E_x]\!]_{\mathcal{T}} \mid [\![E_y]\!]_{\mathcal{T}}$	// Similarly for other operators
$[\![\ (x \text{ in } E)\ ]\!]_{\mathcal{T}}$	$\rightarrow (x \text{ in } [\![E]\!]_{\mathcal{T}})$	
$[\![\text{Ex op } E_y]\!]_{\mathcal{T}}$	$\rightarrow \text{map op } ([\![E_x]\!]_{\mathcal{T}}\ \&\ [\![E_y]\!]_{\mathcal{T}})$	**// Operations, e.g. +**
$[\![\text{op } E]\!]_{\mathcal{T}}$	$\rightarrow \text{map op } ([\![E_x]\!]_{\mathcal{T}})$	
$[\![!E]\!]_{\mathcal{T}}$	$\rightarrow\ ![\![E]\!]_{\mathcal{T}}$	// **Default transforms**
$[\![p]\!]_{\mathcal{T}}$	$\rightarrow p$	// where p is a normalized primary expression
$[\![c\langle t_1,\ldots,t_n\rangle]\!]_{\mathcal{T}} \rightarrow\ c\langle [\![t_1]\!]_{\mathcal{T}},\ldots, [\![t_n]\!]_{\mathcal{T}}\rangle$		// Otherwise homomorphism, where c is composed of terms t_i

Fig. 4. Translation of control constructs and operations into the iterator calculus.

the expression. Blocks with local declarations similarly shift initializers into the sequence body, and are mapped into closures which thus bind the returned generator to the local declarations.

The kernel that implements the iterator calculus also provides a number of built-in methods that optimize several of the most frequently occurring calculus expressions. These include *forall* that is equivalent to reduce with a don't care operator. We also abbreviate *forall*(*x*:1) as *x.bound*(), which represents what Icon calls a bounded expression, that is, a singleton iterator that runs to failure, here optimized as an iterator that always fails but also remembers if it was non-empty. We further optimize the kernel by incorporating direct support for such frequently occurring patterns as *succeed*:1, where *succeed* is *not*(*fail*), as well as *exists*, which is implemented as always restarting the iterator.

Lift primary to iterator

$\llbracket !f(e_1,\ldots,e_n)\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconInvokeIterator({->$\llbracket f\rrbracket_{\mathcal{K}}$ ($\llbracket e_1\rrbracket_{\mathcal{K}}$,…, $\llbracket e_n\rrbracket_{\mathcal{K}}$)})	// Delegates to {->f($t_1,\ldots,t_n$)}()
$\llbracket !c[e]\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconIndexSingleton($\llbracket c\rrbracket_{\mathcal{R}}$, {-> $\llbracket e\rrbracket_{\mathcal{K}}$ })	// Index, updatable reference
$\llbracket !o.x_1. \ldots .x_n\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconFieldSingleton($\llbracket o\rrbracket_{\mathcal{RR}}$, "$x_1$", …, "$x_n$")	// Field reference, updatable
$\llbracket !o.x_1. \ldots .x_n ::f\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconFieldSingleton($\llbracket o.x_1. \ldots .x_n ::f\rrbracket_{\mathcal{R}}$)	// Method reference
$\llbracket !\ C.new(e_1,\ldots,e_n)\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconInvokeIterator({-> new C($\llbracket e_1\rrbracket_{\mathcal{K}}$,…, $\llbracket e_n\rrbracket_{\mathcal{K}}$)})	// Synthetic functions
$\llbracket !\ range(e_1, e_2, e_3)\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconToIterator($\llbracket e_1\rrbracket_{\mathcal{K}}$, $\llbracket e_2\rrbracket_{\mathcal{K}}$, $\llbracket e_3\rrbracket_{\mathcal{K}}$)	// changed to constructs
$\llbracket !l\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconValueSingleton(l)	// Literal, excluding collection literals
$\llbracket !p\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconSingleton($\llbracket p\rrbracket_{\mathcal{R}}$)	// Default singleton iterator over reified primary

Reified primary, with getter and setter

$\llbracket x\rrbracket_{\mathcal{R}}$	$\rightarrow$	x_r, if external reference, new IconVar({->x},{rhs->x=rhs})	// Variable reference
$\llbracket <t>\rrbracket_{\mathcal{R}}$	$\rightarrow$	t_r, if inside closure, t_r.get()	// Temporary
$\llbracket o.x_1. \ldots .x_n\rrbracket_{\mathcal{R}}$	$\rightarrow$	new IconField($\llbracket o\rrbracket_{\mathcal{RR}}$, "$x_1$", …, "$x_n$")	// Field reference
$\llbracket p\rrbracket_{\mathcal{R}}$	$\rightarrow$	{-> $\llbracket p\rrbracket_{\mathcal{K}}$}, if inside closure, $\llbracket p\rrbracket_{\mathcal{K}}$	// Default is closure over term

Read-only reified primary, used in object reference

$\llbracket x\rrbracket_{\mathcal{RR}}$	$\rightarrow$	x_r, if external reference, if inside closure then x else {-> x}	// Variable reference
$\llbracket <t>\rrbracket_{\mathcal{RR}}$	$\rightarrow$	t_r	// Temporary
$\llbracket p\rrbracket_{\mathcal{RR}}$	$\rightarrow$	$\llbracket p\rrbracket_{\mathcal{R}}$	// Default

Inside primary, i.e., is function argument, subscript, or field in object reference

$\llbracket x\rrbracket_{\mathcal{K}}$	$\rightarrow$	x_r.deref(), if external reference or class field then x	// Variable reference
$\llbracket <t>\rrbracket_{\mathcal{K}}$	$\rightarrow$	t_r.deref(), where deref()=get().get()	// Temporary
$\llbracket o.x_1. \ldots .x_n\rrbracket_{\mathcal{K}}$	$\rightarrow$	$\llbracket o\rrbracket_{\mathcal{K}}$ .x_1. … .x_n	// Field reference
$\llbracket o.x_1. \ldots .x_n::f\rrbracket_{\mathcal{K}}$	$\rightarrow$	$\llbracket o\rrbracket_{\mathcal{K}}$.$x_1$. … .$x_n$.f	// Method reference
$\llbracket\ [e_1,\ldots,e_n]\ \rrbracket_{\mathcal{K}}$	$\rightarrow$	[$\llbracket e_1\rrbracket_{\mathcal{K}}$,..., $\llbracket e_n\rrbracket_{\mathcal{K}}$]	// List
$\llbracket\ [e_k^1:e_v^1,\ldots,e_k^n:e_v^n]\ \rrbracket_{\mathcal{K}}$	$\rightarrow$	[$\llbracket e_k^1\rrbracket_{\mathcal{K}}$: $\llbracket e_v^1\rrbracket_{\mathcal{K}}$,..., $\llbracket e_k^n\rrbracket_{\mathcal{K}}$: $\llbracket e_v^n\rrbracket_{\mathcal{K}}$]	// Map
$\llbracket n\rrbracket_{\mathcal{K}}$	$\rightarrow$	nG, where G is Groovy arbitrary precision	// Number
$\llbracket\ l\ \rrbracket_{\mathcal{K}}$	$\rightarrow$	l	// Literal

Calculus operations and control constructs

$\llbracket \text{if } I_1 \text{ then } I_2\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconIf($\llbracket I_1\rrbracket_{\mathcal{K}}$, $\llbracket I_2\rrbracket_{\mathcal{K}}$)	// IconIf encapsulates calculus
$\llbracket (t \text{ in } I)\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconIn($\llbracket t\rrbracket_{\mathcal{R}}$, $\llbracket I\rrbracket_{\mathcal{K}}$)	// Bound iteration
$\llbracket I_1\ \&\ I_2\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconProduct($\llbracket I_1\rrbracket_{\mathcal{K}}$, $\llbracket I_2\rrbracket_{\mathcal{K}}$)	// Similarly for other operators
$\llbracket fail\rrbracket_{\mathcal{K}}$	$\rightarrow$	new IconFail()	
$\llbracket p\rrbracket_{\mathcal{K}}$	$\rightarrow$	p	// Default transform

where x and y are identifiers, t is a temporary identifier, p is a normalized primary, f, c, and e are simple normalized primaries, o is a simple identifier or literal or $< t >$, l is a literal, n is a number, and x_r = new IconVar({-> x}, {rhs->x=rhs}), i.e., a property with get() and set(rhs) methods.
An identifier is an external reference if it is not a class field, method local, parameter, or temporary.

Fig. 5. Concretization of iterator calculus terms.

As mentioned previously, the normalization transform $\mathcal{N}$ for primary expressions is independent of the transform $\mathcal{T}$ for larger expressions, so as to enable staging them separately. The net transform for taking Unicon expressions into the iterator calculus is thus the composition:

$$\mathcal{T} \circ \mathcal{N}$$

and the overall transformation that takes Unicon expressions into Groovy is the composition:

$$\mathcal{K} \circ \mathcal{T} \circ \mathcal{N}$$

where $\mathcal{K}$ concretizes the calculus into Groovy. Figure 5 shows the transforms in $\mathcal{K}$ that take normalized terms in the calculus into Groovy. In particular, variable declarations are exposed as both plain definitions as well as a reified reference to the variable in the form of a property with a getter and setter. Lifted variables are then converted to singleton iterators over their reification, and so return updatable references, while lifted literals are singletons that return immutable values. Similarly, indexing as well as field reference is captured as a singleton iterator that freezes its values on iteration and returns an updatable reference. In contrast, function invocation is captured as a closure and passed to an IconIterator subclass that at the start of iteration invokes the closure, and either delegates iteration to its returned generator, or for a native Java method simply promotes its result to a singleton iterator.

The concretization transform $\mathcal{K}$ is optimized to avoid redundant nested closure formation, for example inside method arguments, and to recognize final reified temporaries that do not need to be inside closures at all. While it might seem easier to have instead made all variables and fields just be reified data types, by carefully exposing class fields and methods as plain definitions, as well as leaving data types and method invocations in native format, we achieve clean integration with Java, and can freely reference Java classes and fields from Junicon and vice versa.

3.4 Translation of classes and methods

Having transformed control constructs and expressions into the iterator calculus, the remaining transforms rely on a relatively straightforward mapping that takes Unicon methods into variadic lambda expressions, and that maps Icon procedures and global variables onto the Java class model using static fields in a class of the same name. We denote the transformations for classes and methods, as well as procedures and globals, by $\mathcal{C}$. Since normalization is a separately staged transformation, and since $\mathcal{T}$ and $\mathcal{K}$ are bundled into $\mathcal{C}$, the net transform that takes programs into Groovy is thus

$$\mathcal{C} \circ \mathcal{N}$$

The Unicon class model has a few differences from Java that must be accommodated in translation. Overall Unicon classes are roughly similar to those of Java, in that they are composed of fields and methods, albeit with relaxed typing and with the exception that multiple inheritance is allowed. However, each method can take any number of arguments, i.e., it is variadic. Method arguments may also be omitted in invocation even if they are interior, e.g., f(x,,y), in which case they are null or resolve to parameter defaults, as well as superfluously supplied, in which case they are ignored. Each method thus has arbitrary arity, which precludes method overloading, and as a consequence each method has a unique name within its class. Unicon also has a scoping model in which variables that are not declared and are unresolved at link time are made local to a method or procedure. While the transforms faithfully preserve other features in Unicon, the linking model in its deferred scoping of locals is an exception, due to its incompatibility with Java. We thus deprecate Unicon's treatment of undeclared variables as local, since this cannot be determined at compile time.

```
⟦global G⟧𝒞                       →   class G { static def  G;                              // Globals
                                      }  import static G.G;

⟦procedure P(x) {                  →   class P {                                             // Procedures
      local z:=e; body                       static def  P = ⟦{(x) -> local z:=e; body}⟧ℒ
}⟧𝒞                                   }  import static P.P;

⟦class C:E (field) {               →   class C extends E {                                   // Classes
                                             def  field;                             // Constructor fields
                                             def  field_r = new IconVar({->field}, {rhs->field=rhs});
                                             C() { super();  initially() }           // Constructors
                                             C(field) { super();  this.field=field;  initially() }
                                             static def  C = ⟦{(x) -> new C(x)}⟧𝒦
      local i:=e;                            def  i=⟦e⟧𝒦∘𝒯.next();                  // Class fields
                                             def  i_r = new IconVar({->i}, {rhs->i=rhs});
      method M(x) { local z:=e; body }       def  M = ⟦{(x) -> local z:=e; body}⟧ℒ   // Method closures
      initially () { local z:=e'; body' }    def  initially = ⟦{-> local z:=e'; body'}⟧ℒ   // Initializer
}⟧𝒞                                   }  import static C.C;

⟦ {(x,y) -> local z:=e; body}⟧ℒ   →   ⟦ {(x,y) -> local z;  ⟦{z:=e;  body;  fail}⟧𝒯 }⟧𝒦    // Lambdas

⟦ {(x, y=d, o[]) -> local z; body𝒯}⟧𝒦   →   { Object...  args ->              // Lambda concretization
                  def  x;   def  x_r = new IconVar({->x}, {rhs->x=rhs});          // Parameters
                  def  y;   def  y_r = new IconVar({->y}, {rhs->y=rhs});
                  def  o;   def  o_r = new IconVar({->o}, {rhs->o=rhs});
                  def  z;   def  z_r = new IconVar({->z}, {rhs->z=rhs});          // Locals
                  if (args == null) { args = omit.getEmptyArray(); };             // Unpack arguments
                  x = (args.length > 1) ? args[1] : null;                           // Can omit argument
                  y = ((args.length > 2) && (args[2] != omit)) ? args[2] : d;       // Has default
                  o = (args.length > 3) ?                                           // Remainder as list
                       Arrays.asList(args).subList(3,args.length) :new ArrayList();
                  return ⟦body𝒯⟧𝒦;          }
```

Fig. 6. Transformation of classes and methods using variadic lambda expressions.

In contrast to Unicon, Junicon follows the Java package model and its scoping rules, so that all variables must be imported or declared. Junicon also slightly extends Unicon syntax to allow a familiar Java-like block notation for classes and methods that uses braces instead of "end" and that uses semicolons instead of newlines to terminate statements. While either Unicon-style or Java-style notation is allowed as input to the interpreter, a preprocessing step aligns programs with the Java-like syntax before transformation. For simplicity, the examples and transformation rules that follow use the Java-style block notation that exists after preprocessing.

A key problem in the translation of Unicon to Groovy is how to cleanly map Unicon procedures and global variables into the Java class model. In particular, procedures, which are akin to methods bound to a global variable, are updatable entities. In addition we need to support Unicon's provision of function-like class instantiation using *C(x)* instead of *new C(x)*, a style similar to that later adopted by Python. A technique that provides a common solution to the above problems is to uniformly map Unicon procedures, global variables, and class constructors into the Java package model by making them static fields in a class with the same name.

These fields can then be bound to variadic lambda expressions to effect methods and constructors. For example, a global G is simply mapped into

    class G { static def G; }

and made visible in scope using

    import static G.G;

Procedures, such as procedure P(x) {body}, are similarly transformed to:

class P { static def P = $[\![\{(x) \text{ -> body}\}]\!]_{\mathcal{L}}$ }

where $\mathcal{L}$ transforms a lambda expression into Groovy. Lastly, support for function-like class instantiation, e.g. *C*(*x*) instead of *new C*(*x*), is realized using static fields that wrap the original constructor:

class C { static def C = $[\![\{(x) \text{ -> new C(x)}\}]\!]_{\mathcal{K}}$ }

Figure 6 illustrates the above transformations for globals and procedures, as well as for methods and classes. The class declaration proper translates fairly directly to a Groovy class, with the caveat that multiple inheritance is realized using Groovy mixins. Local variable declarations within classes are preserved as fields using *def* ; however, variable initializers, having been transformed to generators, must be unraveled after transformation using *next*() so as to yield a value for assignment. Seamless integration of Junicon with Java is made possible by exposing public class fields as normal Java values, and separately encoding their reification.

Methods as well as procedures are implemented using variadic lambda expressions, in a manner that supports argument omission and parameter defaults. The treatment of methods as fields bound to such lambda expressions, or parameterized closures in Groovy, also enables their use as references in generator expressions. Recall that method invocations were normalized to iteration over a returned generator as follows:

e(e',e'') $\rightarrow$ (f in e) & (x in e') & (y in e'') & (o in !f(x,y))

Method definitions must thus return an iterator, and so are transformed to parameterized closures as follows:

method M(x) { local z:=e; body } $\rightarrow$ def M = $[\![\{(x) \text{ -> local z:=e; body}\}]\!]_{\mathcal{L}}$

and from there into a generator function via $\mathcal{L}$:

def M = $[\![\ \{(x) \text{ -> local z; } [\![\{\text{z:=e; body; fail}\}]\!]_{\mathcal{T}}\}]\!]_{\mathcal{K}}$

The transform $\mathcal{K}$ then concretely takes lambda expressions such as those above into variadic lambda expressions that support argument omission, as shown in the bottom of Figure 6. As with block declarations, initializers in local declarations of lambda expressions must be incorporated by $\mathcal{L}$ into the sequence body before further transformation, since initializers in general are also generator expressions. Lambda expressions as well as block declarations must also synthesize local declarations for temporaries used in bound iterators. Method invocation is further optimized by caching freed method body iterators and then reusing them, to avoid unnecessary reallocation of expression trees in the method body.

In interactive mode, the transforms slightly alter their behavior to enable execution of outermost expressions. As with class field initializers, a simple expression such as:

```
class C(lower)              # Original program
   method printRange (upto)
      local i
      every (i := C().range(lower,upto)) do
            System.out::println(i)
   end
   method range (from,bound) … end
end

class C(lower) {            # After preprocessing
  method printRange (upto) {
      local i;
      every (i := C().range(lower,upto)) do
            System.out::println(i);
  }
  method range (from,bound) { … }
}
```

```
class C(lower) {            # Normalized program
   method printRange (upto) {
      local x_0;
      local i;
      every (!i := (x_0 in !C()) &
               !x_0.deref().range(lower, upto)) do
            !System.out::println(i);
      !null
   }
   method range (from,bound) { ... }
}
```

Fig. 7. Preprocessing and normalization of a sample program.

f(x)

is transformed to:

$[\![f(x)]\!]_{\mathcal{K}\circ\mathcal{T}}$.next()

which executes the first iteration of the generator. Moreover, to make constructors for interactively defined classes visible, class definitions have

import static C.C;

appended to their transformation. Lastly, local declarations that are interactive and outermost are transformed in the same manner as class fields, with the exception that they have "def" stripped in order to make them top-level script bindings under Groovy.

There are several other ancillary but simple transformations needed to complete the Unicon translation, for example to enforce arbitrary precision arithmetic. As further described in Section 5, the transformations are implemented in two phases of XSLT transformation: a normalization stage $\mathcal{N}$, followed by $\mathcal{C}$ for the transformation of classes, methods, and expressions into Groovy.

Figures 7 and 8 show the results of preprocessing, normalization, and transformation for an example program. The program in Figure 7 generates the values between 1 and an upper bound, and interfaces with the Java *println* method to effect printing. In Figure 7, the original Unicon program is shown on the top left, and the program after preprocessing and normalization are shown on the bottom left and right, respectively. The Groovy result after the transformation $\mathcal{C}$ is then shown in Figure 8.

Preprocessing, in addition to handling directives for conditional compilation and source inclusion, also inserts semicolons and braces to align programs with a Java-style block notation that simplifies the recognition of parseable statements. Normalization, as shown in the right of Figure 7, then flattens nested generators into products of bound iterators, and indicates where lifting must occur.

The transformation of classes, as shown in Figure 8, can be seen to take methods into variadic lambda expressions assigned to a class field with the original method name. The function body itself is an iterator constructor, so that the function when

```
class C {
   private def methodCache = new MethodBodyCache();       // Method body cache
   public def lower;                                       // Constructor fields and their reification
   private IconVar lower_r = new IconVar({-> lower}, {rhs -> lower=rhs});
   public C() { ; }                                        // Constructors
   public C(lower) { this.lower = lower; }
   public static def C = { Object... args ->               // Static variadic constructor
      if (args ==  null) { args = IconEnum.getEmptyArray(); };
      return new C((args.length > 0) ? args[0] : null);
   };
   public def printRange = { Object... args ->             // Methods
      IconIterator body = methodCache.getFree("printRange");        // Reuse method body
      if (body != null) { return body.reset().unpackArgs(args); };
      def upto;                                                     // Parameters, and their reification
      def upto_r = new IconVar({-> upto}, {rhs -> upto=rhs}).local();
      def x_0;                                                      // Temporaries
      def x_0_r = new IconVar({-> x_0}, {rhs -> x_0=rhs});
      def i;                                                        // Locals, and their reification
      def i_r = new IconVar({-> i}, {rhs -> i=rhs}).local();
      def unpack = { Object... params ->                            // Unpack parameters
         if (params ==  null) { params = IconEnum.getEmptyArray();}
         upto = (params.length > 0) ? params[0] : null;
         i = null;                                                  // Reset locals
      };
      // Method body
      body = new IconSequence(new IconEvery((new IconAssign().over(new IconSingleton(i_r),
         new IconProduct(new IconIn(x_0_r, new IconInvokeIterator({-> C()})),
         new IconInvokeIterator({-> x_0.deref().range(lower, upto)})))),
         new IconInvokeIterator({-> System.out.println(i)})),
         new IconValueSingleton(null), new IconFail());
      // Return body after unpacking arguments
      body.setCache(methodCache, "printRange");
      body.setUnpackClosure(unpack).unpackArgs(args);
      return body;
   }
   def range = { Object... args -> ... }
}
import static C.C;
```

Fig. 8. Transformation of the sample program to Groovy.

invoked will return an iterator; for optimization the iterator body is cached in a stack upon method return, and then reused. The provision of function-like constructors is made possible using static fields with the same name as the class, that cut through to normal constructors, and that are brought into scope using "import static".

Closures, i.e., lambda expressions, and iterators can thus be seen to be the building blocks of the implementation, used both to realize class methods as well as the compact kernel that implements suspendable generators and their composition.

4. TRANSLATION TO JAVA

It is possible to translate Junicon into Java bytecode using the Groovy compiler on the transformed program for use outside the interpreter. However, the motivations of improved performance as well as the removal of dependencies on external Groovy libraries argue for examining the feasibility of direct translation of Junicon into Java. Such a migration to Java also provides insight into the stability of the normalization and transformation algorithms under retargeting. The retargeting of the transforms must address key differences of Groovy from Java, including in particular the use of typed declarations as well as differences in closure and lambda expression notation and behavior. In particular, under Java, forward references are not allowed inside lambda expressions, and all references to method local variables must be effectively final.

The above differences can be addressed with only minor changes to the concretization transformations and class generation, and notably no changes to normalization. Translation to Java is enabled by simply aligning the syntax for closures to that of Java lambda expressions, and by slightly modifying the concretization transforms to emit types, e.g., *Object* instead of *def*. For classes, only a slight modification to method definition and invocation is required to expose methods as variadic lambda expressions. However, these modifications must carefully overcome limitations in forward references as well as subtle differences in the syntax for invoking lambda expressions.

The problem of forward references is handled by defining methods normally albeit with variadic parameters, and then also exposing them as method references assigned to fields with the same name. In Java, method references are lambda expressions for methods that already have a name, denoted by "o::f". Under our scheme, the dual method references are given priority over the corresponding method names in resolution, accommodating Unicon reference semantics, while the plainly defined methods allow forward references to be used. The above technique has the added benefit of promoting seamless integration with Java, in that external Java code can invoke Junicon methods as just methods rather than as lambda expressions. Lastly, it is straightforward to ensure that method locals are effectively final by encapsulating them as a reified variable that holds its own value.

However, a key challenge that must be addressed is the different way that Java treats invocation using lambda expressions in comparison to Groovy closures. Recall that in Unicon methods are first class citizens, i.e., they can be passed in expressions, and so must be exposed as references in some form. In Groovy such a reference takes the form of a closure, and there is no syntactic difference in invoking a method from a closure. Invocation of a closure under Groovy transparently uses the same notation as if it were a method, e.g., "f(x)". However, in Java invocation using lambda expressions does not use the same syntax as for method invocation, as it might otherwise if function types had been introduced. Rather, in Java, a lambda expression resolves to an instance that implements an interface with a single method, called a functional interface. Invocation using lambda expressions, or variables that hold lambda expressions, are explicitly differentiated from method invocation in that they must use a field name, for example "f.apply(args)" instead of "f(args)". The above difference must thus be incorporated into the concretization transforms for method invocation. Invocation must explicitly accommodate a lambda expression, and so after normalization is translated as follows:

f(x) $\rightarrow$ ((VariadicFunction) f).apply(x)

We do make the simple optimization that, if an invocation refers to a method within the immediate class, then: f(x) $\rightarrow$ f(x). However, it turns out the performance savings for this are minimal.

Figure 9 summarizes the changes needed to retarget the transformations from Groovy to Java, while Figure 10 shows the example program from Figure 7 after translation to Java. In addition to treating methods as lambdas, there are a few other subtleties, for example changing collection literals such as [1,2,3] as well as numeric

Method transform changes for Java

⟦ method M(x) { local z:=e; body }⟧$_{\mathcal{C}}$ → public Object M = (VariadicFunction) this::M;
public Object M ⟦(x) { local z:=e; body }⟧$_{\mathcal{M}}$

⟦ (x,y) { local z:=e; body }⟧$_{\mathcal{M}}$ → ⟦ (x,y) { local z; ⟦{z:=e; body; fail}⟧$_{\mathcal{T}}$ }⟧$_{\mathcal{K}}$

⟦ (x, y=d, o[]) {local z; body$_{\mathcal{T}}$}⟧$_{\mathcal{K}}$ → (Object... args) { **// Method concretization**

```
IconVar  x_r  = new IconVar();                              // Parameters
IconVar  y_r = new IconVar();                               // Final reified variable
IconVar  o_r = new IconVar();                               //    holds its own value
IconVar  z_r = new IconVar();   ;                           // Locals
if (args == null) { args = omit.getEmptyArray(); };         // Unpack parameters
x_r.set((args.length > 1) ? args[1] : null);                // Can omit argument
y_r.set (((args.length > 2) && (args[2] != omit)) ? args[2] : d);  // Has default
o_r.set((args.length > 3) ?                                 // Remainder as list
        Arrays.asList(args).subList(3,args.length) :new ArrayList());
return ⟦body_T⟧_K;          }
```

⟦ {(x, y=d, o[]) -> local z; body$_{\mathcal{T}}$}⟧$_{\mathcal{K}}$ → { Object... args -> **// Lambda concretization**
Same as method concretization
}

Concretization changes for Java

⟦! f(e$_1$,…,e$_n$)⟧$_{\mathcal{K}}$ → new IconInvokeIterator(( ) -> ((VariadicFunction) ⟦f⟧$_{\mathcal{K}}$).apply (⟦e$_1$⟧$_{\mathcal{K}}$,…, ⟦e$_n$⟧$_{\mathcal{K}}$)})

⟦! o.x$_1$. … .x$_n$::f(e$_1$,…,e$_n$)⟧$_{\mathcal{K}}$ → new IconInvokeIterator(( ) -> ⟦o⟧$_{\mathcal{K}}$.x$_1$. … .x$_n$.f (⟦e$_1$⟧$_{\mathcal{K}}$,…, ⟦e$_n$⟧$_{\mathcal{K}}$))

⟦o.x$_1$. … .x$_n$ ⟧$_{\mathcal{K}}$ → IconField.getFieldValue(⟦o⟧$_{\mathcal{RR}}$, "x$_1$", …, "x$_n$") // Only if o is not typed

⟦o.x$_1$. … .x$_n$::f ⟧$_{\mathcal{K}}$ → ⟦o⟧$_{\mathcal{K}}$.x$_1$. … .x$_n$::f // Stays method reference

⟦ [e$_1$,...,e$_n$] ⟧$_{\mathcal{K}}$ → new IconList(⟦e$_1$⟧$_{\mathcal{K}}$,..., ⟦e$_n$⟧$_{\mathcal{K}}$) // List collection literal

⟦ [e$_k^1$:e$_v^1$,...,e$_k^n$:e$_v^n$] ⟧$_{\mathcal{K}}$ → new IconMap(⟦e$_k^1$⟧$_{\mathcal{K}}$,⟦e$_v^1$⟧$_{\mathcal{K}}$,..., ⟦e$_k^n$⟧$_{\mathcal{K}}$,⟦e$_v^n$⟧$_{\mathcal{K}}$) // Map collection literal

⟦n⟧$_{\mathcal{K}}$ → new IconNumber.IconInteger(n), or IconDecimal(n) // Configurable arbitrary precision

where methods are exposed as method references using the
interface VariadicFunction <T,R> { R apply (T... args); }

Fig. 9. Transformation of methods and their invocation to Java.

literals to method invocations for list formation and arbitrary precision promotion, respectively. Moreover, any explicitly typed variables must be carefully carried forward into derived reified variables and method parameters, so as to enable field reference without reflection when needed, as well as their use in typed Java methods. The above changes to the transforms, as well as those described below, are parameterized in the XSLT transforms of the implementation, so that the interpreter can generate either Groovy or Java code.

Another problem that must be addressed is how to differentiate the invocation of native Java methods from other Unicon methods, given that the latter invocation is assumed to use a functional interface. In the general case the function name used in an invocation may refer either to a Unicon variable or method, or to a Java method in external code. In the former case the name resolves to a lambda expression, while in the latter case the name resolves to a plain method. One solution to differentiate the invocation of lambda expressions from that of plain methods is to make explicit the use of non-Unicon Java methods. Invocation of plain Java methods uses an explicit notation that is then translated to a field reference that is assumed to be typed, e.g.,

```
public class C {
   private MethodBodyCache methodCache = new MethodBodyCache();   // Method body cache
   public Object printRange = (VariadicFunction) this::printRange;          // Method references
   public Object range = (VariadicFunction) this::range;
   public Object lower;                                                      // Constructor fields
   private IconVar lower_r = new IconVar(()-> lower, (rhs)-> lower=rhs);
   public C() { ; }                                                          // Constructors
   public C(Object lower) { this.lower = lower; }
   public static VariadicFunction C= (Object... args) -> {                  // Static variadic constructor
      if (args ==  null) { args  = IconEnum.getEmptyArray(); };
      return new C((args.length > 0) ? args[0] : null);
   };
   public Object printRange (Object... args) {                              // Methods
      IconIterator body = methodCache.getFree("printRange");  // Reuse method body
      if (body != null) { return body.reset().unpackArgs(args); };
      IconVar upto_r = new IconVar().local();                  // Parameters
      IconTmp x_0_r = new IconTmp();                            // Temporaries
      IconVar i_r = new IconVar().local();                      // Locals
      VariadicFunction unpack = (Object... params) -> {         // Unpack parameters
         if (params ==  null) { params = IconEnum.getEmptyArray(); };
         upto_r.set((params.length > 0) ? params[0] : null);
         i_r.set(null);                                         // Reset locals
         return null;
      };
      // Method body
      body = new IconSequence(new IconEvery((new IconAssign().over(new IconSingleton(i_r),
        new IconProduct(new IconIn(x_0_r, new IconInvokeIterator(()-> ((VariadicFunction) C).apply())),
        new IconInvokeIterator(()-> ((VariadicFunction) IconField.getFieldValue(
              x_0_r, "range")).apply(lower, upto_r.deref())))))),
        new IconInvokeIterator(()-> System.out.println(i_r.deref()))),
        new IconValueSingleton(null), new IconFail());
      // Return body after unpacking arguments
      body.setCache(methodCache, "printRange");
      body.setUnpackClosure(unpack).unpackArgs(args);
      return body;
   }
   public Object range (Object... args) { ... }
}
```

Fig. 10. Translation of the sample program to Java.

o::f(x) → o.f(x)

The above translation works when targeting either Groovy or Java. Otherwise, invocation is assumed to use a variadic lambda expression.

The alternative to making Java invocation explicit is to scope up through superclass and import definitions to resolve whether external references are to Java or Unicon, since they treat method invocation differently. While such a technique is feasible, our strategy has been to purposefully avoid such resolution, since it replicates many details of the Java compiler for handling types and mutual dependencies. Such added complex resolution techniques must also be maintained to track the evolving Java type system, and so the risk outweighs the minor benefits.

Lastly, another migration path to Java that was explored was to replace the use of lambda expressions with inner classes. The above required extremely few changes to the concrete transformation, since closure and method reference generation was already encapsulated. The concrete transformations then simply translate:

(args)->{body} → new VariadicFunction() { public (Object... args) { return body; } }
this::m → new VariadicFunction() { public (Object... args) { return m(args); } }

While targeting inner classes does allow the use of versions of Java before Java 8, the performance impact was minimal, and so the default for the interpreter is to use the more succinct lambda expressions.

5. USING XSLT FOR PROGRAM TRANSFORMATION

There are a broad array of transformation tools that could be brought to bear to implement the above rewriting rules [Feather 1987; Visser 2005]. These technologies range from program transformation systems such as Stratego/XT [Bravenboer et al. 2008] and Spoofax [Kats and Visser 2010], to metaprogramming support in languages such as Groovy [Dearle 2010]. While the former are more formally based on concepts from term writing and theorem proving, the latter provides more ad-hoc support for manipulating the exposed syntax tree within the language itself, for example in Groovy to provide annotations for aspect-oriented techniques such as mixin classes. However, our goal is not to provide such dynamic support for syntax extension, nor in the interests of retargetability do we wish to be too heavily bound to a dependency such as Groovy. At the same time, the simplicity of the transforms for Unicon do not demand the power, scope, or formality of full-fledged transformation tools such as those above.

5.1 XSLT-based transformation

As part of this research we explore the utility of using XSLT as an alternative means of transformation. XSLT is a language for transforming XML documents expressed in XML itself [Kay 2008; Clark 1999; Clark and DeRose 1999]. An XSLT transform consists of a set of templates, or production rules, whose preconditions are XPath patterns and that substitute the specified content for any matched XML node. The production rules can be grouped into modes as well as prioritized to effect specific rewriting strategies. For example, the XSLT templates for taking Unicon into Groovy, i.e., the transformation rules, are partitioned into modes for each of the transforms $\mathcal{N}$, $\boldsymbol{\mathcal{F}}$ within $\mathcal{N}$, $\boldsymbol{\mathcal{L}}$, and $\mathcal{C}$.

For illustration, two rewrite rules that create local declarations for all temporary variables in bound iterators within a given block scope are shown in Figure 11. The XSLT templates in Figure 11, in an extremely succinct fashion, only create local variable definitions for temporaries that appear within a block but not in any subordinate block, i.e., if the temporary and block have the same block ancestor count. We have found XSLT to be remarkably expressive at similarly capturing expression context, e.g., looking up or down in scope for class or variable declarations that are referenced in an expression, which makes it quite an effective tool for transformation. While XSLT was found to be effective in this small-scale scenario – the transformations for all of Junicon are less than 3800 lines – its verbosity and lack of formal basis may hinder its scalable application to other domains. On the other hand, XSLT is standardized and its Version 1.0, which we use, is built into the Java runtime environment. For our purposes, which is migration between high-level languages rather than incorporating metaprogramming support, it was found to be highly advantageous.

```
<xsl:template match="BLOCK"  mode="findTemporaries"  priority="2">
        <xsl:copy>
                <xsl:copy-of select="@*"/>
                <xsl:variable name="blockDepth" select="count(ancestor-or-self::BLOCK)"/>
                <xsl:apply-templates select=".//EXPRESSION[@isTemporary and
                        ($blockDepth = count(ancestor::BLOCK))]"  mode="createLocal"/>
                <xsl:copy-of select="*"/>
        </xsl:copy>
</xsl:template>

<xsl:template match="*"  mode="createLocal"  priority="2">
        <xsl:param name="variableName" select="@tmpVariableName"/>
        <STATEMENT>
                <KEYWORD>def</KEYWORD>
                <DECLARATION>
                        <IDENTIFIER>
                              <xsl:value-of select="$variableName "/>
                        </IDENTIFIER>
                </DECLARATION>
                <DELIMITER ID=";"/>
        </STATEMENT>
</xsl:template>
```

Fig. 11. XSLT template for synthesizing temporary variable declarations.

5.2 Structure of the transformational interpreter

The transformation of Unicon programs and their execution on the Groovy substrate are housed within a generic harness for transformational interpretation. The transformational interpreter supports multi-stage transformations that are not necessarily tied to XSLT, multiple pluggable execution substrates, and cross-correlation of error messages back to the original source. The steps involved in transformation are broken down into an end-of-statement detector that uses a chain of preprocessors, followed by a parser into a decorated XML syntax tree, then filtering, normalization, and translation transforms, and lastly either dispatching the transformed expression to another transformational interpreter for further processing, or deconstructing and piping it into a substrate scripting engine for execution. The interpreter is coded in Java, and uses Spring dependency injection [Walls 2011] as well as the Java scripting API to enable customization of the above steps as well as injecting scripting engine substrates. The generic harness can function either as an interactive line-by-line interpreter, which is no small feat for Unicon given its Pascal-like syntax and multi-line string literals, or as a tool that emits transformed code that can then be compiled for example into Java bytecode.

The Junicon interpreter is an instantiation of the above harness, customized with a Unicon preprocessor in Java, a Javacc LL(k) grammar [Reis 2011] that conservatively extends Unicon syntax and that emits decorated XML abstract syntax trees, parameterized XSLT transforms for normalization and translation to either Groovy or Java, and a Java kernel that implements the iterator calculus. These four pieces of customization amount to roughly 7000 lines of code. Each component is carefully engineered to be capable of being run standalone. Together they fully define the transformation of Unicon into both Groovy and Java.

Interpreter behavior can be rapidly customized through the XSLT templates for transformation, as well as a Groovy prelude that allows extensions to the interpreter kernel. Such scripted customization of the interpreter enables rapid prototyping of translation enhancements within a spiral development methodology. Indeed, the Junicon kernel was initially prototyped in Groovy, and later refined into Java to provide improved performance.

6. PERFORMANCE

Reasonable performance, comparable to that of Unicon, is an important goal. While the concerns of Java integration, compactness, and semantic clarity are paramount, an implementation with a slowdown on several orders of magnitude would be of no practical use. Although extreme measures for optimization have not been taken, obviously wasteful implementation techniques, such as redundant reified declarations and repeated iterator construction in method bodies, have been avoided. To evaluate the practical viability of the implementation, measurements of both compiled Groovy and compiled Java translations were undertaken, and compared to that natively run under Unicon.

The performance of Junicon relative to that of Unicon was benchmarked using a suite of six programs. The programs exercise a wide range of Unicon features: "Matrix Multiply" employs list creation and access and is $O(n^3)$, "Quick Sort" exercises recursive method invocation and is $O(n \log(n))$, "New Instances" exercises instance creation and field access and is $O(n)$, "Pi Digits", which computes pi to a given length, exercises arbitrary precision arithmetic and loop iteration, and is $O(n^2)$, "Loop Test" which is $O(n^2)$ measures the basic efficiency of loop constructs and the iterator calculus, and "Suspend Test", which is $O(n)$ and structured to be similar to that of the "Loop Test", measures the overhead of suspend and resume in method invocation. Sample sizes for each program were chosen to uniformly effect exponential execution time, ranging from 2 seconds to one hour, and each sample point is the average of three runs.

Figures 12 and 13 show the performance of Junicon relative to that of Unicon for compiled Groovy and compiled Java translations, respectively. The performance comparisons in Figure 12 are based on code compiled under Groovy 2.3.1 that exploits Java invoke dynamic support, and run under Java 1.8.0. In contrast, Figure 13 shows the relative performance of Junicon translated to Java using lambda expressions, and then compiled and run under Java 1.8.0. The benchmarks were run on an AMD Dual-Core Opteron 2212 with 8GB of memory running Linux Mint 12. The execution times were measured using the "time" command by adding both the user and system time, the latter which includes the overhead of Java Just-in-Time (JIT) compilation running on the second core.

The results show that Junicon yields only marginally worse, and sometimes better, performance than that of Unicon. In Figures 12 and 13, each data point represents the ratio of Junicon's execution time to that of Unicon for a given program and sample size. Values below 1 on the y-axis demonstrate better performance than that of Unicon, while values above 1 correspond to worse performance. In both figures, in the initial stages the performance of Junicon rapidly improves over time as Java's Just-in-Time (JIT) dynamic compiler converts often used methods, both user and system, to directly executable instructions.

Both Groovy and Java over the long term have roughly similar performance; however, Groovy’s initial performance is quite a bit slower than that of Java, and as Groovy has a larger dependency set it takes longer for JIT to have full effect. For example, in Figure 12 Quick Sort initially has a performance slowdown over 15 relative to that of Unicon; the inset to Figure 12 shows the full graph with the y-axis

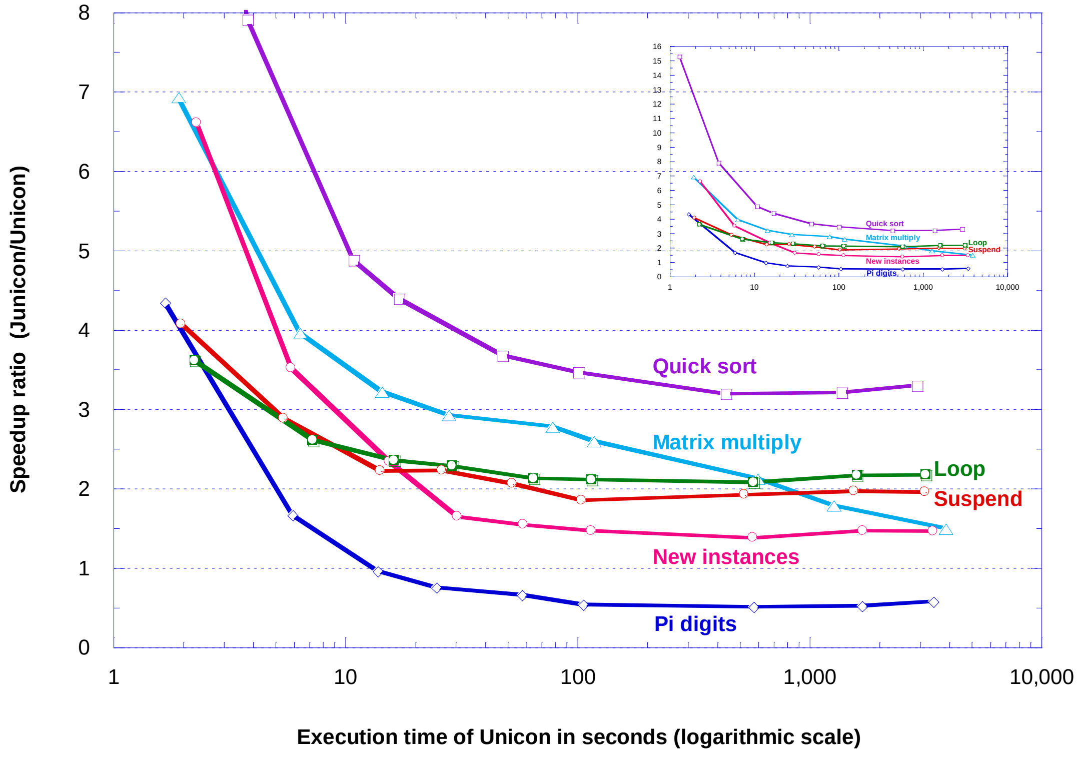

Fig. 12. Performance of Junicon when translated to Groovy.

expanded to include Quick Sort’s first data point. Moreover, Groovy's overhead in dynamic dispatch is evidenced in the degraded performance of "New Instances", which heavily employs static method invocation to overlay instance construction. Surprisingly, as shown in "Pi Digits", the use of Java's arbitrary precision arithmetic performs better than that of Unicon. Lastly, our implementation of suspend is highly optimized and incurs zero additional cost over a normal iterator, as evidenced when comparing the performance of "Loop Test" to the "Suspend Test", which are roughly equivalent programs.

If one views the "Loop Test" as giving a baseline to the overhead of using the iterator calculus to implement generator expressions, which is roughly a factor of 2 slowdown over Unicon, then the other sample programs are similarly impacted by that inherent overhead. Thus any speedup beyond the baseline of a factor of 2 slowdown, representing calculus overhead, may be a more accurate measure of improvement of a given feature’s performance over that of Unicon. Using that analysis, the performance of "Pi Digits" could be interpreted to indicate a speedup by a factor of 4 over Unicon for the feature of arbitrary precision arithmetic. Lastly, we also examined the performance of several other variants such as using inner classes instead of lambda expressions in Java, as well as exposing methods in Groovy using function references in a manner similar to that done for Java, with little difference observed.

Of particular interest is that the performance of Junicon is roughly equivalent, or even a little faster, than that reported for Jcon relative to Icon. As mentioned, Jcon used a different implementation technique based on instrumentation of fail-resume

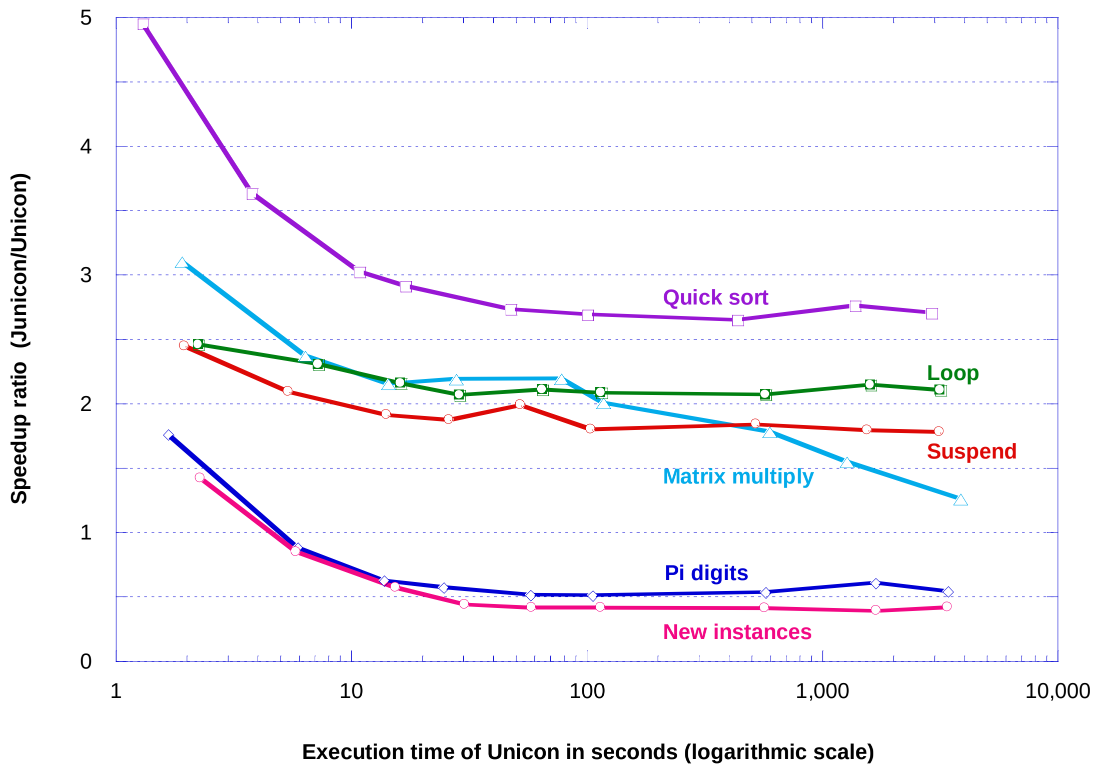

Fig. 13. Performance of Junicon when translated to Java.

ports to produce JVM byte code, and reported an overall slowdown by a factor of two relative to Icon [Proebsting and Townsend 2000]. The roughly similar performance results between the two radically different implementation techniques of Junicon and Jcon, despite that fact that Junicon does not directly generate bytecode, might imply that the performance difference between Unicon and the two Java translations is due to the inherent overhead of translating a dynamic language for generator expressions into Java. Consequently one might be led to surmise that Junicon's technique for transformation into an iterator calculus is potentially as efficient as any other implementation technique.

7. RELATED WORK

The complexity of Unicon's compact notation for goal-directed evaluation has motivated several novel translation techniques and attempts at formally defining its meaning. Chief among these efforts in the arena of Java implementations was Jcon [Proebsting and Townsend 2000], a bytecode generator for Icon that relied on a Prolog-like Byrd-box model [Byrd 1980] to instrument backtracking using fail and resume ports [Proebsting 1997]. Other implementation and semantics studies primarily relied on continuation-based approaches, such as recursive interpretation using failure continuations [O'Bagy and Griswold 1987], cross-compilation into C using a continuation-passing-style [O'Bagy et al. 1993], a denotational semantics based on continuations [Gudeman 1992], and a semantics based on list and continuation monads [Danvy et al. 2002].

Monads in particular are potentially a natural fit for capturing Icon semantics. List monads in Haskell, with bind as concatenation over map, can effect lazy list comprehension, while the Maybe monad and the monad fail method captures failure [Bird 1998; Hudak et al. 1999; Jones 2003]. List and continuation monad semantics for a small subset of Icon were examined in [Danvy et al. 2002], with compilation by semantics-driven partial evaluation. The residual programs of the above continuation-based approach instrument code with suspend and resume advice. Jcon similarly relied on heavy instrumentation of code and data types, which rendered problematic its interfacing with other Java programs. Concerns also arise for efficiency and unnecessary code complexity. Haskell might give for example a 30-times performance decrease over procedural Java for a given algorithm. It bears noting that while the aforementioned denotational and monad semantics for Icon [Gudeman 1992; Danvy et al. 2002] were not incorrect, they were a bit incomplete in not explicitly addressing method application, which involves another dimension of iteration if the method name is an expression. While the "to" construct was addressed as a prototypical generator function, i.e., non-monogenic operator [Gudeman 1992], the general case of application using method expressions was not, and has particular implications in implementation, in that method references or closures, i.e. lambda abstraction, are needed in the translation target. Nor did the above efforts directly address the object-oriented extensions provided by Unicon and its impact on propagating generator expressions through fields in object references.

In contrast to these efforts our approach is one of exposing implicit generators in a more recognizable explicit form that is aligned with native invocation mechanisms and that maintains consistency with Java iterators. Our equational formulation of control constructs similarly illuminates their meaning in a simpler manner. Our approach differs from other ways to implement iterator abstractions using continuations, threads, or higher-order functions, in that we take a purely iterator-based view that rewrites nested generators, and so rely on flattening rather than instrumentation or higher-order functions to do that work. Aligning with the target substrate is key both to enable the grafting of goal-directed capabilities onto another language, for example through scoped annotations that delimit iterator propagation, and to enable an interactive interpreter, a feature that was to date lacking for Unicon. Another advantage of our approach is that certain problematic features of Icon and Unicon, such as first-class patterns [Walker 1989] and concurrency [Gharaibeh et al. 2012a; Gharaibeh 2012b] potentially become simpler to implement.

Our extension of Java iterators to support suspendable iteration bears some similarity to other work on interruptible iterators in JMatch [Liu et al. 2006; Liu and Myers 2003]. There the focus was on extending coroutine iterators in JMatch to handle update operations on the underlying data structure. Interruptable iterators for ML were also examined by Filliatre [2005] using purely functional persistent cursors that also allow backtracking. In contrast, our iterators integrate suspend and resume with failure-driven control as well as compositions such as product, concatenation, reduce, and map, 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. However, the cost of multithreading is not minor, and as we already translate programs to iterator expressions, it is simpler to directly augment iterators with suspension. Lastly, our iterator calculus has many aggregate operations similar to the Stream interface in Java that, in conjunction with lambda expressions, supports a functional programming style. However, unlike streams, the calculus operations are suspendable and failure-driven, have no terminal operations that yield non-iterators, and add CSP-like guarded choice and repetition needed for composing generators.

There are several more formal program transformation systems that could be used instead of XSLT [Feather 1987; Visser 2005]. These include such tools as Stratego/XT [Bravenboer et al. 2008]. While such tools could be brought to bear to effect translation in this case, the motivation of our approach was to examine the utility of XSLT as an alternative for small-scale program transformation across dynamic languages. XSLT has the potential advantage of being a widespread standard that represents a trade off of simplicity for reduced dependency on more complicated tools.

While it is well understood that the semantics of Icon generators can be built on lazy list comprehension, to our knowledge no one has formalized a calculus for specifying such comprehensions directly. Unlike typical mechanisms for list comprehension, the iterator calculus allows specifying comprehensions using first-order formulae similar to those found in Z schemata [Abrial et al. 1980; Spivey 1992; Davies and Woodcock 1996] and SETL [Dewar et al. 1981; Schwartz et al. 1986].

8. CONCLUSIONS

We have developed a Java-based interpreter for Unicon, called Junicon, implemented using XSLT-based program transformation. Such transformational interpretation has several distinct advantages. First, the implementation clarifies the semantics of Unicon by reducing nested generators to a familiar and explicit form, and yields an equational definition of its control constructs. Second, the normalization techniques that make explicit the otherwise implicit generator propagation enable the grafting of goal-directed evaluation onto other languages such as Java. Third, by translating Unicon onto another Java-based language, and by carefully preserving native types and invocation mechanisms, the implementation can cleanly integrate with and leverage the full range of Java capabilities including its portability and libraries for concurrency and graphics.

Lastly, the Junicon implementation demonstrates that XSLT is potentially well suited for expressing several types of program transformations. While not terribly well founded in formal methods, the efficacy of XSLT as a rewriting system and its built-in support in Java make it extremely attractive. Moreover, its use yielded an extremely compact implementation that is readily extensible and retargetable. Rapid development of translation enhancements is facilitated by the scripted nature of the XSLT transforms. We examined the ease of retargeting by migrating the transforms to Java using its nascent support of lambda abstraction, and so realized an interpreter that can function either interactively or as an offline translator.

Currently the prototype implementation [Mills and Jeffery 2014], while transforming the full Unicon syntax, maps only a core subset of the vast array of Unicon operators and built-in functions into a Java implementation. Future efforts will focus on extending Junicon to implement the full range of Unicon capabilities. We also plan to investigate the possibility of automatically porting adjunct Unicon libraries for networking, graphics, and external language integration using facilities such as JNI and Swig. Lastly, correlating debugging and performance information within a transformational framework is an area to be further explored.

ACKNOWLEDGMENTS

The authors would like to thank in advance the anonymous reviewers for their helpful suggestions and comments on drafts of this article.

REFERENCES

Abrial, J.-R., Schuman, S. A., and Meyer, B. 1980. A specification language. In *On the Construction of Programs*, A. M. Macnaghten and R. M. McKeag. Eds., Cambridge University Press, Chapter 11, 343-410.

Allison, L. 1990. Continuations implement generators and streams. *The Computer Journal 33*, 5, 460-465.

Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM 21*, 8, 613-641.

Bird, R. 1998. *Introduction to Functional Programming using Haskell (2nd Edition).* Prentice Hall Press. http://www.cs.ox.ac.uk/publications/books/functional/.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program. 72,* 1, 52-70.

Byrd, L. 1980. Understanding the control of Prolog programs. Technical Report 151. Department of Artificial Intelligence, University of Edinburgh, Scotland. 12 pages.

Clark, J. (Ed.). 1999. XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 (November 1999). http://www.w3.org/TR/xslt

Clark, J. and DeRose S. (Eds.). 1999. XML Path Language (XPath), Version 1.0. W3C Recommendation 16 (November 1999). http://www.w3.org/TR/xpath

Davies, J. and Woodcock, J. 1996. *Using Z: Specification, Refinement and Proof.* Prentice Hall International Series in Computer Science. http://www.usingz.com/text/online/.

Dearle, F. 2010. *Groovy for Domain-Specific Languages.* Packt Publishing. http://groovy.codehaus.org/

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

Dewar, R. B. K., Schwartz, J. T., and Schonberg, E. 1981. Higher level programming: Introduction to the use of the set-theoretic programming language SETL. Technical Report, Computer Science Department, Courant Institute of Mathematical Sciences, New York University.

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy, and formal derivation of programs. *Comm. ACM 18*, 8, 453-457.

Feather, M. S. 1987. A survey and classification of some program transformation approaches and techniques. In *IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation.* North-Holland Publishing Co. Amsterdam, The Netherlands, 165-195.

Filliatre J.-C. 2006. Backtracking iterators. In *Proceedings of the 2006 Workshop on ML (ML'06).* ACM Press, New York, NY, 55-62.

Gharaibeh, J. A. 2012. *Programming Language Support for Virtual Environments.* Ph.D. Dissertation. University of Idaho, Moscow, Idaho.

Gharaibeh, J.A., Jeffery, C., and Oikonomou, K.N. 2012. An hybrid model for very high level threads. In *Proceedings of the 2012 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2012).* ACM Press, New York, NY, 55-63.

Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. 2014. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional.

Griswold, R. and Griswold, M. 1996. *The Icon Programming Language, Third Edition.* Peer-to-Peer Communications.

Griswold, R.E., Poage, J. F., and Polonsky, I. P. 1971. *The SNOBOL 4 Programming Language, 2nd Edition.* Prentice-Hall, Englewood Cliffs, N.J.

Griswold, R. E., Hanson, D. R., and Korb, J. T. 1981. Generators in Icon. *ACM Trans. Program. Lang. Syst. 3*, 2, 144-161.

Gudeman, D. A. 1992. Denotational semantics of a goal-directed language. *ACM Trans. Program. Lang. Syst. 14*, 1, 107-125.

Hoare, C.A.R. 1978. Communicating sequential processes, *Comm. ACM 21*, 8, 666-677.

Hudak, P., Peterson, J., and Fasel, J. 1999. A gentle introduction to Haskell 98. Technical Report, Yale University, 64 pages.

Jagger, J., Perry, N., and Sestoft, P. 2007. *C# Annotated Standard.* Morgan Kaufmann Publishers Inc., San Francisco, CA.

Jeffery, C. L. 2001. Goal-directed object-oriented programming in Unicon. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC'01).* ACM Press, New York, NY, 306-308.

Jeffery, C., Mohamed, S., and Gharaibeh, J. A. 2013. Unicon language reference. Technical Report UTR8a. University of Idaho, Moscow, Idaho. 47 pages. http://www.unicon.sourceforge.net/utr/utr8a.pdf

Jeffery, C., Mohamed, S., Gharaibeh, J. A., Pereda, R., and Parlett, R. 2013. *Programming with Unicon, 2nd Edition.* http://www.unicon.sourceforge.net/ub/ub.pdf

Jones, S. P. (Ed.). 2003. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press. http://haskell.org/onlinereport/.

Jones, S. P. and Wadler, P. 1993. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93).* ACM Press, New York, NY, 71-84.

Kay, M. 2008. *XSLT 2.0 and XPath 2.0 Programmer's Reference, 4th Edition.* Wrox Press, New York, NY.

Lennart, C. L., Kats, L. C. L., and Visser, E. 2010. The spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10).* ACM Press, New York, NY, 444-463.

Li, G. 2010. *Formal verification of Programs and Their Transformations.* Ph.D. Disseration. University of Utah, Salt Lake City, Utah.

Liang, S., Hudak, P., and Jones, M. P. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95).* ACM Press, New York, NY, 333-343.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1977. Abstraction mechanisms in CLU. *Comm. ACM 20,* 8, 564-576.

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J. C., Scheifler, R., and Snyder, A. 1981. CLU reference manual. *Lecture Notes in Computer Science 114,* G. Goos and J. Hartmanis, Eds., Springer-Verlag, Berlin.

Liu, J. and Myers, A. C. 2003. JMatch: Iterable abstract pattern matching for Java. In *Proceedings 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03).* Lecture Notes in Computer Science 2562, Springer-Verlag, 110-127.

Liu, J., Kimball, A., and Myers, A. C. 2006. Interruptible iterators. In *Proceedings 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming (POPL'06).* ACM Press, New York, NY, 283-294.

Mills, P. and Jeffery, C. 2014. The Junicon project at sourceforge. http://sourceforge.net/projects/junicon/.

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

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, 217-239.

Proebsting, T. A. 1997. Simple translation of goal-directed evaluation. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97).* ACM Press, New York, NY, 1-6.

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

Reddy, U. S. 1990. Formal methods in transformational derivation of programs. In *Proceedings Formal methods in software development.* ACM Press, New York, NY, 104-114.

Reis, A. J. D. 2011. *Compiler Construction Using Java, JavaCC, and Yacc.* Wiley-IEEE Computer Society Press.

Rossum, G. and Drake, F. L. 2011. *The Python Language Reference Manual.* Network Theory Ltd., Godalming, United Kingdom.

Schwartz, J. T., Dewar, R. B., Schonberg, E., and Dubinsky E. 1986. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, NY.

Spivey, J. M. 1992. *The Z Notation: A Reference Manual, 2nd Edition.* Prentice Hall International Series in Computer Science, Upper Saddle River, NJ. http://spivey.oriel.ox.ac.uk/mike/zrm/.

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, 831-873.

Wadler, P. 1990. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming.* ACM Press, New York, NY, 61-78.

Walker, K. W. 1989. First-class patterns for Icon. *Computer Languages 14*, 3, 153-163.

Walls, C. 2011. *Spring in Action, 3rd Edition.* Manning Publications Co., Greenwich, CT.