Simple Translation of Goal-Directed Evaluation

Todd A. Proebsting*
The University of Arizona

Abstract

This paper presents a simple, powerful and flexible technique for reasoning about and translating the goal-directed evaluation of programming language constructs that either succeed (and generate sequences of values) or fail. The technique generalizes the *Byrd Box*, a well-known device for describing Prolog backtracking.

1 Motivation

In the current world of programming language development, an enormous amount of effort is going into developing new ways of expressing and manipulating data values (e.g., type theory, object-oriented theory, etc.) and very little effort is going towards incorporating richer control-flow constructs into modern languages. As evidence, note that CLU-style iterators have been well-understood for around 20 years [LSAS77] and yet they appear in no mainstream language.[1]

Generators (iterators) and goal-directed expression evaluation are extremely powerful control-flow mechanisms for succinctly expressing operations that operate over a sequence of values. The Prolog programming language derives much of its power from goal-directed evaluation (i.e., backtracking) in combination with unification [Byr80]. The Icon programming language is an expression-oriented language that combines generators and goal-directed evaluation into a powerful control-flow mechanism [GG83].

One possible explanation for the slow adoption of generators and goal-directed evaluation into mainstream languages may be the perceived difficulty of implementing them correctly and efficiently. This papers presents a new technique for implementing goal-directed evaluation of expressions that generate a sequence of values. The technique is simple, understandable, and yields efficient code.

2 Icon Introduction

I will use the Icon programming language as a basis for explaining the new translation scheme, although the translation scheme is applicable to other goal-directed languages.

All Icon expressions *succeed* in generating zero or more values. An expression that cannot produce any more values *fails*. For example, the expression

```
1 to 5
```

generates the values 1, 2, 3, 4, 5, and then fails.

Combining expressions with operators or function calls creates a compound expression that combines all subexpression values and generates all possible result values prior to failing. The expression

```
(1 to 3) * (1 to 2)
```

generates the values 1, 2, 2, 4, 3, 6, and then fails. Subexpressions evaluate left-to-right—the previous sequence represents 1×1, 1×2, 2×1, 2×2, 3×1, 3×2. Note that the right-hand expression is re-evaluated for each value generated by the left-hand expression.

Generators may have generators as subexpressions. The expression

```
(1 to 2) to (2 to 3)
```

generates 1, 2, 1, 2, 3, 2, 2, 3, and then fails. Those values are produced because the outer (middle) `to` generator is actually initiated four times: `1 to 2`, `1 to 3`, `2 to 2`, and `2 to 3`.

*Address: Todd A. Proebsting, Department of Computer Science, University of Arizona, Tucson, AZ 85721; Telephone: 520/621-4326; Email: todd@cs.arizona.edu. http://www.cs.arizona.edu/people/todd/

[1] It's a shame iterators were not adopted by the Java designers — Java hype seems to have revived garbage collection and might have done the same for iterators.

Icon's expression evaluation mechanism is goal-directed. Goal-directed evaluation forces expressions to re-evaluate subexpressions as necessary to produce as many values as possible. To demonstrate this, we introduce Icon's relational operator **<**. The **<** operator takes two numeric operands and returns the value of the right operand if it is greater than the value of the left, otherwise, it fails (and, therefore, generates no value). Goal-directed evaluation forces **<** to re-evaluate its operand expressions as necessary to produce values on which it will succeed. The expression

```
2 < (1 to 4)
```

generates the values 3, 4, and then fails. Similarly,

```
3 < ((1 to 3) * (1 to 2))
```

generates 4, 6, and then fails.

Generators and goal-directed evaluation combine to create succinct programs with implicit control flow.

3 Byrd Box

Like Icon, Prolog evaluates programs in a goal-oriented fashion. Unlike Icon, Prolog uses unification and backtracking to produce a sequence of substitutions. Nonetheless, their goal-directed evaluation mechanisms are similar in that expressions ("calls" in Prolog) are started, succeed or fail, and may be resumed.

Byrd [Byr80] concisely summarized the execution of Prolog clauses by describing control-flow changes between pairs of calls via four ports:[2]

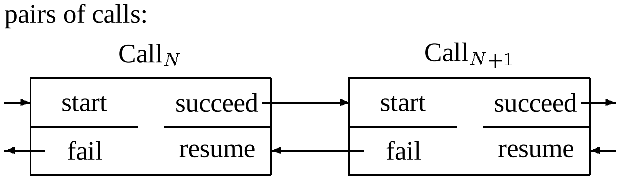
start The *start* port is the initial entry point into the evaluation of a particular call.

resume The *resume* port is the subsequent re-entry point for all re-evaluations of a particular call.

fail The *fail* port is the departure point from a call that has just failed.

succeed The *succeed* port is the departure point for all successful satisfactions of a particular call.

For each call, Byrd constructed a *box* that consisted of these four program points. Combining the boxes in sequence models the backtracking control flow between pairs of calls:



[2] Byrd called these *call*, *redo*, *fail*, and *exit*.

Satisfying one call leads directly to the initial invocation of a subordinate call. Similarly, the failure of a call causes the re-evaluation of the invoking call.

Finkel and Solomon [FS80, Fin96] independently developed a similar four-port model of control flow. They used it to describe the control flow of *power loops*. Power loops backtrack and thus the start/succeed/resume/fail model describes their behavior well. Unlike Prolog, however, power loops cannot be described by a simple sequential connection of four-port boxes.

4 New Technique

The four-port technique of describing backtracking control flow is the basis for my technique of describing the control flow of generators and goal-directed evaluation. This new technique generalizes Byrd's model and allows the "boxes" to be combined in ways that are more powerful than Byrd's simple linear model—similar in some respects to the Finkel and Solomon model.[3] Unlike any previous uses of the four-port model, the new technique describes control-flow constructs that require making some of the connections between ports at run-time.

This translation technique is syntax-directed. For each operator in a program's abstract syntax tree (AST), translation produces four labeled chunks of code—one for each of Byrd's ports. In addition, each AST operator has a corresponding run-time temporary variable to hold the values it computes. Thus, the translation will produce four code chunks for each operator, θ:

θ**.start** The initial code executed for the entire expression rooted at θ.

θ**.resume** The code executed for resuming the expression rooted at θ.

θ**.fail** The code executed when the expression rooted at θ fails.

θ**.succeed** The code executed when the expression rooted at θ succeeds at producing a value.

The specification of these code chunks is similar to the specification of attribute grammars, except that nothing is actually computed. Instead, each code chunk is specified by a simple template. The start and resume chunks are synthesized attributes. The fail and succeed chunks are inherited attributes. Having both inherited and synthesized chunks allows control to be threaded arbitrarily among an

[3] I learned of power loops in a class from Prof. Finkel in 1984 at the University of Wisconsin. Undoubtedly, I got the basic idea of a four-port translation scheme in that class, although I *thought* I was inventing it from scratch. To the best of my knowledge, the generalizations of the four-port method are my own.

operator and its children, which is necessary for some goal-directed operations. In the Byrd Box, ports are locations, whereas here they are pieces of code.

The evaluation of some Icon operators requires additional temporary variables and code chunks.

4.1 Translating "N"

Possibly the simplest expression to translate is a single numeric literal (e.g., "3"). A numeric literal represents a sequence of length one. The code for a numeric literal will immediately produce its value and exit. Upon resumption, it will fail. Note that the code chunks for handling success and failure are "inherited" from an enclosing expression, and therefore cannot be specified here.

literal$_N$

literal$_N$.start	:	**literal**$_N$.*value* $\leftarrow N$
	:	`goto` **literal**$_N$.succeed
literal$_N$.resume	:	`goto` **literal**$_N$.fail

4.2 Translating Unary Operators

Mathematical unary operators such as negation are also easy to translate, and they give a simple idea of how succeed and fail chunks are created. Starting and resuming the negation expression requires starting and resuming its subexpression. Negating an expression is straightforward: for each value the subexpression generates, simply negate that value and succeed; fail when the subexpression fails.

uminus(E)

uminus.start	:	`goto` E.start
uminus.resume	:	`goto` E.resume
E.fail	:	`goto` **uminus**.fail
E.succeed	:	**uminus**.*value* $\leftarrow -E$.*value*
	:	`goto` **uminus**.succeed

4.3 Translating Binary Addition

Binary operators introduce the first interesting threading of control among the various code chunks. Translating $E_1 + E_2$ requires that all values of E_2 be produced for *each* value of E_1 and that the sums of those values be generated in order. Thus, resuming the addition initiates a resumption of E_2, and E_1 is resumed when E_2 fails to produce another result. Starting the addition expression requires that E_1 be started, and for each value E_1 generates, E_2 must be (re-)started (*not* resumed). The addition fails when E_1 can no longer produce results. The following specification captures the semantics cleanly.

plus(E_1, E_2)

plus.start	:	`goto` E_1.start
plus.resume	:	`goto` E_2.resume
E_1.fail	:	`goto` **plus**.fail
E_1.succeed	:	`goto` E_2.start
E_2.fail	:	`goto` E_1.resume
E_2.succeed	:	**plus**.*value* $\leftarrow E_1$.*value*+E_2.*value*
	:	`goto` **plus**.succeed

Unlike addition, a relational operator (e.g., `>`, `=`, etc.) may fail to produce a value after its subexpressions succeed. When a comparison fails, it resumes execution of its right operand in order to have other subexpressions to compare (i.e., it is goal-directed, and seeks success):

LessThan(E_1, E_2)

LessThan.start	:	`goto` E_1.start
LessThan.resume	:	`goto` E_2.resume
E_1.fail	:	`goto` **LessThan**.fail
E_1.succeed	:	`goto` E_2.start
E_2.fail	:	`goto` E_1.resume
E_2.succeed	:	`if` (E_1.*value* $\geq$ E_2.*value*) `goto` E_2.resume
	:	**LessThan**.*value* $\leftarrow E_2$.*value*
	:	`goto` **LessThan**.succeed

4.4 Translating Builtin Generators

Builtin operations, like "E_1 `to` E_2", are equally easy to translate in this framework. The "`to`" generator produces every integer from E_1 to E_2 in ascending order. Furthermore, it must generate those values for every pair of values that E_1 and E_2 produce. The code below uses an extra code chunk as well as an additional temporary variable.

to(E_1, E_2)

to.start	:	`goto` E_1.start
to.resume	:	**to**.$I \leftarrow$ **to**.I + 1
	:	`goto` **to**.code
E_1.fail	:	`goto` **to**.fail
E_1.succeed	:	`goto` E_2.start
E_2.fail	:	`goto` E_1.resume
E_2.succeed	:	**to**.$I \leftarrow E_1$.*value*
	:	`goto` **to**.code
to.code	:	`if` (**to**.I $>$ E_2.*value*) `goto` E_2.resume
	:	**to**.*value* $\leftarrow$ **to**.I
	:	`goto` **to**.succeed

4.5 Translating Conditional Control-Flow

The previous translations used direct gotos to connect various chunks in a fixed fashion at compile time. For some operations this is not possible. The `if` expression,

`if` E_1 `then` E_2 `else` E_3

Evaluates E_1 exactly once, simply to determine if it succeeds or fails. If E_1 succeeds then the `if` expression generates the E_2 sequence (and fails when E_2 fails), otherwise the `if` generates the E_3 sequence until failure.

Translating an `if` statement into the four-port model requires defering the `if`'s resumption action until runtime. If E_1 succeeds, then the `if`'s resume action must be to resume E_2. Otherwise, the `if`'s resume action is to resume E_3. This translates into an *indirect* `goto` based on a temporary value, "*gate*." E_1's succeed and fail chunks set *gate* to the appropriate chunk's—either E_1's or E_2's—resume label.

ifstmt(E_1, E_2, E_3)

ifstmt.start	:	`goto` E_1.start
ifstmt.resume	:	`goto` [**ifstmt**.*gate*]
E_1.fail	:	**ifstmt**.*gate* ← `addrOf` E_3.resume
	:	`goto` E_3.start
E_1.succeed	:	**ifstmt**.*gate* ← `addrOf` E_2.resume
	:	`goto` E_2.start
E_2.fail	:	`goto` **ifstmt**.fail
E_2.succeed	:	**ifstmt**.*value* ← E_2.*value*
	:	`goto` **ifstmt**.succeed
E_3.fail	:	`goto` **ifstmt**.fail
E_3.succeed	:	**ifstmt**.*value* ← E_3.*value*
	:	`goto` **ifstmt**.succeed

4.6 Translating Other Operations

This new four-port model is capable of succinctly describing every type of Icon operator, including loops, conditionals, and function calls. The previous examples include all the necessary parts (i.e., `goto`'s, indirect `goto`'s, and simple computations) for building the code chunks. Translating a function call that generates a sequence of values requires a mechanism for suspending and resuming a function invocation.

5 Example Translation

Translating Icon expressions in a syntax-directed fashion with these four-port templates is easy. For instance, the translation of

`5` $>$ `((1 to`$_1$ `2) * (3 to`$_2$ `4))`

requires expanding the templates for 1, 2, 3, 4, 5, `*`, `to`$_1$, `to`$_2$, and `>`. Figure 1 gives all of the code chunks for the nine expanded templates.

The example demonstrates that while the technique is simple, it suffers from generating many simple copies and many branches to branches. Propagating copies and eliminating branches to branches (by branch chaining and reordering the code) optimizes the code well. Figure 2 gives the result of performing these optimizations on the code in Figure 1. The result closely resembles code that would be produced from two generic `for` loops, which is exactly what one would hope for.

6 Related Work

Independently, Byrd, and Finkel and Solomon developed a four-port model for describing backtracking control flow—see Section 3 for more details. It is not clear if Byrd invented the four-port box for translation purposes, or for debugging purposes [Byr80]. It appears that Byrd used the boxes to model control flow between calls within a single clause, but not to model the flow of control between clauses within a procedure, nor to model the control-flow in and out of a procedure. Finkel and Solomon used their four-port scheme to describe power loops. In neither case was the idea of four-ports generalized into a mechanism for describing how four pieces of code might be generated and stitched together for various operators in a goal-directed language.

Many people have studied the translation of Icon's goal-directed evaluation. The popular Icon translation system, which translates Icon into a bytecode for interpretation, controls goal-directed evaluation by maintaining a stack of *generator* frames that indicate, among other things, what action should be taken upon failure [GG86]. Special bytecodes act to manipulate this stack—by pushing, popping or modifying generator frames—to achieve the desired goal-directed behavior. The new scheme requires nothing more powerful than conditional, direct, and indirect jumps.

O'Bagy and Griswold developed a technique for translating Icon that utilized *recursive interpreters* [OG87]. The basic idea behind recursive interpreters for goal-directed evaluation is that each generator that produces a value does so by recursively invoking the interpreter. Doing so preserves (*suspends*) the generator's state for possible resumption when the just-invoked interpreter returns. A recursively invoked interpreter's return value indicates whether the suspended generator should resume or fail. O'Bagy's interpreter executes the same bytecode as the original Icon interpreter. Recursive interpreters suffer from the overhead of recursive function calls.

Gudeman developed a goal-directed evaluation mechanism that uses *continuation-passing* to direct control flow

Label	**Code**	**Label**	**Code**
1.start	1.*value* $\leftarrow 1$ goto 1.succeed	1.resume	goto 1.fail
2.start	2.*value* $\leftarrow 2$ goto 2.succeed	2.resume	goto 2.fail
3.start	3.*value* $\leftarrow 3$ goto 3.succeed	3.resume	goto 3.fail
4.start	4.*value* $\leftarrow 4$ goto 4.succeed	4.resume	goto 4.fail
5.start	5.*value* $\leftarrow 5$ goto 5.succeed	5.resume	goto 5.fail
mult.start	goto to$_1$.start	mult.resume	goto to$_2$.resume
to$_1$.fail	goto mult.fail	to$_1$.succeed	goto to$_2$.start
to$_2$.fail	goto to$_1$.resume	to$_2$.succeed	mult.*value* $\leftarrow$ to$_1$.*value* * to$_2$.*value* goto mult.succeed
to$_1$.start	goto 1.start	to$_1$.resume	to$_1$.I $\leftarrow$ to$_1$.I + 1 goto to$_1$.code
1.fail	goto to$_1$.fail	1.succeed	goto 2.start
2.fail	goto 1.resume	2.succeed	to$_1$.I $\leftarrow$ 1.*value* goto to$_1$.code
to$_1$.code	if (to$_1$.I$>$ 2.*value*) goto 2.resume to$_1$.*value* $\leftarrow$ to$_1$.I goto to$_1$.succeed		
to$_2$.start	goto 3.start	to$_2$.resume	to$_2$.I $\leftarrow$ to$_2$.I + 1 goto to$_2$.code
3.fail	goto to$_2$.fail	3.succeed	goto 4.start
4.fail	goto 3.resume	4.succeed	to$_2$.I $\leftarrow$ 3.*value* goto to$_2$.code
to$_2$.code	if (to$_2$.I$>$ 4.*value*) goto 4.resume to$_2$.*value* $\leftarrow$ to$_2$.I goto to$_2$.succeed		
greater.start	goto 5.start	greater.resume	goto mult.resume
5.fail	goto greater.fail	5.succeed	goto mult.start
mult.fail	goto 5.resume	mult.succeed	if (5.*value*$\leq$ mult.*value*) goto mult.resume greater.*value* $\leftarrow$ mult.*value* goto greater.succeed

Figure 1: Templates for “5 $>$ ((1 to$_1$ 2) * (3 to$_2$ 4))”

greater.start	to$_1$.I $\leftarrow$ 1 goto to$_1$.code
to$_1$.resume	to$_1$.I $\leftarrow$ to$_1$.I + 1
to$_1$.code	if (to$_1$.I$>$ 2) goto greater.fail to$_2$.I $\leftarrow$ 3 goto to$_2$.code
greater.resume	to$_2$.I $\leftarrow$ to$_2$.I + 1

to$_2$.code	if (to$_2$.I$>$ 4) goto to$_1$.resume
	mult.*value* $\leftarrow$ to$_1$.I * to$_2$.I
	if (5 $\leq$ mult.*value*) goto greater.resume
	greater.*value* $\leftarrow$ mult.*value*
	goto greater.succeed

Figure 2: Optimized Code for “5 $>$ ((1 to$_1$ 2) * (3 to$_2$ 4))”

[Gud92]. Different continuations for failure and success are maintained for each generator. While continuations can be compiled into efficient code they are notoriously difficult to understand, and few target languages directly support them.

Walker developed an Icon-to-C translator, which used a mechanism very similar to the interpreter's for controlling goal-directed evaluation. This translator concentrated its efforts on data-flow optimizations rather than control-flow optimizations.

7 Conclusion and Future Work

These new techniques will be the basis for a new Icon compiler that will translate Icon to Java bytecodes. The translation of an Icon program's abstract syntax tree will be a simple expansion of its operators, based entirely upon templates like those given previously. After generating code naively, copy propagation and branch elimination will optimize the code. This code generation method is simple to implement and generates efficient code.

8 Acknowledgments

Saumya Debray helped debug my explanation of Prolog control flow—any remaining errors are certainly my fault.

This research is partially supported by grants from IBM Corp., ATT Foundation, NSF (CCR-9502397, CCR-9415932), and ARPA (N66001-96-C-8518, DABJ-63-85-C-0075).

References

[Byr80] Lawrence Byrd. Understanding the control of prolog programs. Technical Report 151, University of Edinburgh, 1980.

[Fin96] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley Publishing Company, 1996. ISBN 0-8053-1192-0.

[FS80] Raphael Finkel and Marvin Solomon. Nested iterators and recursive backtracking. Technical Report 388, University of Wisconsin-Madison, June 1980.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Inc., 1983. ISBN 0-13-449777-5.

[GG86] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, 1986. ISBN 0-691-08431-9.

[Gud92] David A. Gudeman. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Languages and Systems*, 14(1):107–125, January 1992.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, (8):564–576, August 1977.

[OG87] Janalee O'Bagy and Ralph E. Griswold. A recursive interpreter for the icon programming language. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 138–149, St. Paul, Minnesota, June 1987.