G : A Language Based on Demand Driven Evaluation of Streams

Timothy A. Budd John Placer Department of Computer Science Oregon State University Corvallis, Oregon 97330

budd@ cs.orst.edu placer@ cs.orst.edu

The language G is an expression based language designed around the concept of the stream; where a stream is a sequence of values, perhaps infinite in length. There are no statements in G, only expressions. All expressions are evaluated on a demand driven basis, no computation takes place that does not lead directly to a necessary result. In this paper we describe the basic features of G, and show how many currently popular language paradigms can be expressed naturally and easily using the notion of streams.

Introduction

Many programming problems can be expressed naturally in terms of a sequence of values being produced by some computational processes. Such sequences are called *streams*. The language G uses the stream as its basic data structure; every value in G is a stream. Even scalar values such as numbers are interpreted as a stream of length one. Streams are evaluated in G in a strictly demand driven fashion. No value is generated that is not required immediately in some computation. This demand driven nature is extended even to functions. Although it has the apperance of an imperative language, G is actually a declarative language. If we take sequential execution interrupted by control flow directives to be the hallmark of statements in a conventional language, then there are no statements in G, only expressions. Control flow constructs, such as **if** and **while** forms, are treated merely as a means to form expressions. As with all expressions, functions are executed in a demand driven fashion, producing values only when necessary and producing only as many values as are needed to satisfy an immediate request. While programming with streams is unconventional, it is notable that many of the currently popular programming paradigms, such as functional programming [. backus fp .], set programming [. dubinsky setl .], and logic programming [. mellish .], can be expressed quite naturally as extensions to programming with streams. Thus G can truly be said to be a multiparadigm programming language [. hailpern multiparadigm .]. In this paper we describe some of the basic features of the language G and show how the language can be used in a variety of ways. The language G is not the first language to utilize streams; we have studied (and hopefully learned from) a large number of previous languages during the design process for G. These languages include Icon [.icon prhall.], Seque [.griswold seque .], Scheme [.sussman.], Smalltalk [.goldberg blue.], and a host of others. Demand driven implementations of programming languages have also been studied in the context of other languages, including Lisp [. cons wise .] and APL [. budd apl book .].

Constructing Streams

Tuples The basic technique for constructing a stream in G is the tuple. A tuple has various forms, the easist of which is a simple list of elements: [1, 2, 3, 4, 5, 6] Tuples can include any value, and the values need not be of the same type. The following tuple includes an integer, a floating point number, and another tuple. The nested tuple has a string for one of its fields. [1, [2, "abc"], 3.1415926] Tuples can also include various shorthand ways for denoting collections of values. One of these shorthands is the range expression. A range expression describes a sequence of numeric values in arithmetic progression. The following table shows various types of range expressions: [n .. m] The values between n and m, inclusive [n ..] The infinite list of values starting at n [n .. m step k] A difference other than one [n .. step k] Infinite list with non-one step A range may have no legal value, in which case no values are produced. Two or more range expressions can be combined in a single tuple. The following tuple, for example, will enumerate six values: [1 .. 3 , 7 .. 3 , 60 .. 81 step 7] Range expressions are not themselves streams, but are rather stream constructors. They have meaning only within tuple brackets. The other major class of stream constructors in G are code bodies, which will be discussed in a later section. Tuples, and streams in general, are viewed in many different ways in G. A tuple of fixed length is often used as a record; for example it could be used to encode the name, department, and salary of a particular individual in a company: ["fred jones", 4, 43000] A collection of tuples might represent a database, where each entry in the database is a record. For example, a tuple might encode parent-child information for a group of people: parent := [["fred","jennifier"],["fred","mike"], ["mike","sarah"],["jennifer","mary"]] Other times a tuple, or a stream, particularly if it is generated on the fly as necessary, might represent a succession of answers to a particular query. In subsequent sections we will see various ways that tuples and streams can be formed and used.

Basic Operations on Streams

There are several basic operations in G that can be applied to all streams. More complex actions are constructed by building on these fundamental processes. The meaning of any expression in G can be given by describing how the expression will respond to these operations, just as in object oriented languages the meaning of any object is completely given by describing now the object will respond to the set of messages it understands [. budd smalltalk book .]. Letting S represent some stream, these basic operations can be described as follows: Refresh the stream S. Subsequent applications of the @ and * operations (below), will result in the values enumerated by S starting over from the beginning. Streams are automatically refreshed when they are created. Advance the stream and return the next generated value. If the stream has never produced a value this yields the first value. The application of this operator is the basic driving force in producing action in the G system, as the computation of all values is deferred as long as possible. It is only when an explicit request for a value is made that actions, such as function execution, may have to take place. Return the last (the ‘‘current’’) value generated by the stream S. If S has yet to produce a value an implicit @ operation is performed, and the first value enumerated by the stream is returned. Return the number of values that can potentially be enumerated by S. Because of the demand driven semantics of G, this operation may be very costly to execute; while some forms of stream may know their extent, many will have no choice but the enumerate their values, counting the number of elements they produce. It is even possible, in fact common, to have infinite streams. While some streams ‘‘know’’ they are infinite, and will produce a special value in response to a request for their size, other streams that turn out to be infinite because of the nature of their computation, but which cannot be statically determined by the system, may force the system into an infinite loop when asked for their size. Return the index (position in the stream) of the last element generated. Positions are numbered start-

ing at one. If t is a scalar, return the value of the stream at index position t. If t is not a scalar, it must be a stream consisting of two scalar values. These are used as a starting and ending position, and the stream returned is the stream between these two positions. As this operation may be applied to any stream, even a stream being generated by a function, similar comments concerning the possible cost of this operation hold as were noted for the size (#) operation above. Operations on Streams Since everything in G is a stream, all operators must accept streams as input and produce streams as output. There are various different ways that the traditional meanings of operations, such as arithmetic addition and subtraction, relations, and the like could be extended to streams. Take the operation of addition, for example. Given two streams, each enumerating three values, one can argue both for the pointwise addition of elements, yielding a stream of three values, and the all-pairs combination of values, yielding nine numbers: **Point wise evaluation** [1, 2, 3] + [4, 5, 6] = [5, 7, 9]

All pairs evaluation [1, 2, 3] + [4, 5, 6] = [5, 6, 7, 6, 7, 8, 7, 8, 9] There does not seem to be any reason, *a priori*, to favor one interpretation over the other. In these cases our decisions have been guided by asking two questions: Which interpretation seems more natural, and turns out to be most useful in solving problems, and, If we take one interpretation, can the same effect as the other interpretation be easily produced using other mechanisms in the language. Our final decisions on these issues, as well as many other aspects of the language, have not yet been settled. At the moment we are leaning towards a pointwise interpretation on plus, as that has seemed to be most useful. For relationals, on the other hand, the all pairs evaluation seems to be most natural. In addition, we have adopted the Icon convention of having relational operations return the value of their right argument [. icon prhall.]. Thus relational operators can be used as a *filter* [. hanson SL5 .], deleting unwanted values from the enumeration of a stream. If S is a stream, for example, the expression (10 < S) will generate the substream of S representing those elements which are greater than 10. As later examples will illustrate, this often turns out to be extremely convenient. Pattern Matching The last section noted how relational operators can be used to produce a *filter* for streams. A more generalized filtering mechanism is the pattern matching expression. Pattern matching is applicable to streams in which each element generated is itself a stream (most usually a tuple). The pattern matching expression is compared against each element enumerated, and only those elements which match the pattern pass through the filter. Letting S represent a stream, the general form for a pattern matching expression is: S [pattern] The pattern portion is a special tuple that may contain any number of relational expressions in which the first term has been eliminated. For example: S [< 10 , = "abc"] When called upon to produce a value, elements from the underlying stream are examined in turn. Each element which satisfies the pattern; by producing a stream in which the various fields yield a true (that is, nonempty) value, will pass through the filter. Pattern elements can be written without the relational operator, in which case a test for equality is implicitly assumed. A field in the pattern tuple can be left blank to indicate that any value is acceptable. In addition, a pattern matching expression can involve *output variables*. An output variable is simply a variable preceded by a question mark. Such expressions act like a blank field, in that any value is acceptable. As a side effect, however, for each value which passes the filter the corresponding field value is assigned to the output variable. If, for example, **parent** is the database of parent-child information described in the previous section, the expression: parent ["fred" , ?child] Will yield the records in which the string "fred" is the first field. In addition, as each value is enumerated, the variable *child* will be given the value of the second field in the record. In addition to filtering out unwanted values, pattern matching can be used to provide keyed access to databases. If we assume that the second fields in the parent database are unique, then, for example, the query parent [?x , "mary"] Will yield the unique record keyed by the value "mary". Code Bodies In an earlier section we described range expressions. These were not streams themselves, but were syntactic devices for describing streams when used within a tuple expression. Another similar class of syntactic descriptions are code bodies. There are different types of code bodies; the if, the foreach, the repeat, the while, the assignment and the local identifier declaration. We describe each of these in turn. The if form corresponds to if statements in conventional languages. It is written as the keyword **if** followed by an expression in parenthesis followed by a tuple. When called upon to produce a value, it evaluates the expression. If the stream produced by the expression is nonempty (has at least one value), it then yields the values, in turn, produced by the tuple. For example, the expression: [if (1 < 2) [3, 4, 5]] enumerates the three values 3, 4 and 5. If forms can optionally have an **else** clause, which lists the values to be produced should the expression fail to generate at last one value. [if (1 > 2) [3, 4, 5] else [6, 7]] It is often the case that within the tuple following the if one wishes to access the expression tested by the if. This is accomplished using the reserved name **if**, as in the following example: [if (7 > 2) [1, *if, 3]] This stream would generate the values [1, 2, 3]. The foreach form is similar to the if, only the values produced by the underlying tuple are regenerated once for each value produced by the stream given off by the expression. Thus, for example, the expression: [foreach ([3, 4, 5]) [2, *foreach]] Produces a six element sequence, [2, 3, 2, 4, 2, 5]. The repeat form produces either an empty list or an infinite list, by repeatedly looping over the values generated by the underlying tuple. repeat [1, 2, 3] The while form is similar to the if and the foreach. While it is tempting to equate the while form and the foreach, they differ in one important respect; in the while form the expression portion is refreshed and reevaluated each iteration through the loop, while in the foreach form it merely continues producing values. while (i < 3) [1, 2, 3] The assignment has the form identifier := expression The assignment is not executed for its value, since it yields no value. Nevertheless, as a side effect it modifies the current binding of the identifier. If the identifier cannot be matched with any local variable or parameter it is assumed to refer to a global variable. Thus, global variables can be created simply by assigning a value to them. Finally, the local identifier declaration has the form: local [namelist] Like the assignment, this form produces no values, but is evaluated for its side effects. The namelist given in the tuple identifies new local variables which can thereafter be used within the tuple in which the local form appears. In addition, each name can optionally be followed by a colon and an expression denoting an initial value for the variables. The following example shows

local variables x and y being established, with x being given an initial value of 3. [local [x:3, y] , ...] We have purposely avoided calling these forms ‘‘statements’’, since although they look like statements in a conventional language their interpretation is quite different. Most importantly, as with all expressions in G, they are demand driven. They are expressions which produce values only when required, and generate only as many values as necessary to satisfy a given request. They can be refreshed, in which case they will start again from their beginning, or indexed, causing execution to move to an arbitrary location. The examples given in subsequent sections will illustrate some of these features. Functions A function in G serves only to introduce a new name space and to introduce alternative values (arguments) to names of fields in tuples. Thus functions in G are in many ways quite different from functions in conventional languages. The general form of a function in G is the keyword **func**, followed by a parameter expression, followed by a tuple representing the function body. Parameters associated with a specific function invocation can be considered to be a single stream. A keyword identifier, **args**, can be used to access this stream. Thus functions in G can take any number of arguments. The stream of arguments can, however, often be usefully thought of as a fixed length record. The parameter expression is used to introduce names for each of the fields in this record, and to give optional values that will be used should the argument stream not contain corresponding fields. For example, given a definition such as func(A, B:2, C:A+1) [...] The identifier A would be given the value corresponding to the first field in the stream representing the arguments. Should the argument stream have only this one value, the values of B and C would be 2 and the value (A+1), respectively. Should the argument stream have two values, the value of B would be the second field, and the value of C would still be (A+1). Finally, should the argument stream have three or more values, the default values for both B and C would be ignored. The use of a parameter name that can not be bound to a value and has not been given a default value is similar to the use of any uninitialized variable name, and produces a run-time error. The value associated with a function; that is the value that will be produced by a function in response to @ operations, is the value given by the tuple following the parameter expressions, with names bound as just described. Within the tuple associated with a function the name *self* can be used to denote recursive application of the function. Notice there is no ‘‘return’’ statement in G; values are yielded simply by listing them. Thus, for example, a quicksort function can be written as follows: qsort := func(s) [local [t], t := @ s, if (t) [self(t > s), t, self(t <= s)]] Here the function body introduces a new local identifier t, which is assigned the first value yielded by s. If this value is not the empty stream (that is, if s had at least one value), then we return the catenation of the values returned by the function called on the stream consisting of elements less than t, t itself, and the values returned by the function called on the stream consisting of elements greater than or equal to t. Since the *local* form and the assignment form yielded no values, they did not contribute to the result. The if statement, on the other hand, may yield values, which are the values given by the tuple which is part of it. Code forms are not restricted to appearing only in function bodies, and neither are function bodies restricted to using only code forms. The following function definition is entirely legal: silly := func(s) [1, 2, 3, (0<s), 6 .. 9] When presented with a stream S, the value of silly(S) is the stream consisting of the numbers 1, 2 and 3, the values of S which are greater than 0, and the numbers 6, 7, 8 and 9. Problem Solving in G A computer language is not simply a random assemblage of features, but is a vehicle by which users attempt to solve their problems using a computer. The real test of any language is not how simple it is, nor how easy it is to implement, but how easy it is to use in the solution of problems. In this section we consider various classes of problems, and show how the features of G facilitate the solution of problems from these categories. String Processing Our model of string processing is that adopted by Griswold in his language Icon [. icon prhall .]. In fact, we will show how G can be used to perform string processing by showing how many of the string processing features of Icon can be simulated in G. While copying Icon is our starting point for investigating string processing in G, it is probable that as we develop these capabilities further the two languages will slowly diverge. In order to present any nontrivial example requires that we first must build a library of specialized routines. The first routine, **in**, takes two arguments, and returns a nonempty value if the first argument appears in the stream given by the second argument. It relies upon the demand driven semantics of the relational operators. in := func(a, b) [if (a = b) [a]] The second routine, **upto**, takes at least two arguments, and can take as many as four. Upto returns the positions in the second argument of items that can also be found in the first argument (this definition is a direct copy of the built-in function in Icon). The third and forth argument represent the starting and ending locations of the search, and default to the first location and the length of the stream, respectively. upto := func(c, s, start:1, end:#s) [local [t : s![start, end]], foreach (t) [if (in(*foreach, c)) [start+index[foreach]-1]]] The next function, **span**, returns the next position in a stream that has the property that the following element is *not* found in some test stream, returning the last position of the stream if all remaining elements are found in the test stream (this is similar to the Snobol pattern of the same name [. griswold snobol4 .]). span := func(c, s, start:1) [while (in(*s, c))[start := start +1], start - 1]

Using these routines, we can write a simple function which takes a string as input and returns a stream of strings representing the individual words in the input string. Let *chars* be a global variable containing a string representing all letter characters. The function first uses **upto** to find the first letter character, then **span** to find the length of the run of characters, then produces as value the substring between these two points, before updating the starting point for the search for the next character. words := func(s) [local [i:1, j], while (upto(chars, s, i)) [j := span(c, s, *while), s![*while, j], i := j+1]] By suitably defining more support routines other, more extensive, string processing capabilities are also possible [. placer notes .]. Functional Programming The technique of filtering and processing streams of values is naturally applicable to a style of programming in which computation is described by the composition of functions [. backus fp .]. We illustrate this

feature by showing how the **words** function in the previous section could be rewritten in a functional style. We first note that the stream of values returned by **upto** corresponds to positions of characters in the input stream. On this sentence, for example, the stream would be [1,2,3,4,5,6,9,...]. Beginnings of words, therefore, correspond to breaks in continuous runs of integers. We can compute these values using a filter, which we will call **runstarts**. The function creates a separate copy of the parameter string, and compares each element to the next element to be generated: runstarts := func(s) [local [t:s], @ t, foreach(s) [if (*foreach + 1 <> @ t) [*foreach]] Producing the stream of words is therefore a simple matter of finding the start of each word and using **span** to find its extent: words := func(s) [foreach(runstarts(upto(char, s))) [s![*foreach, span(c, s, *foreach)]]] Set Programming Just as we used Icon for our model for string processing, we will use Setl [. dubinsky setl .] as our model for set programming; in particular, our Setl examples will be taken from the interactive dialect of Setl called ISETL [. levin dubinsky .]. The language Setl is noted for its powerful and succinct representation of sets and the ease with which they can be manipulated. To be honest, our representation for the subset of sets that Setl can deal with is not as succinct as the ISETL syntax; where they write: [expression | var in set : predicate] We must make the relationship between the three parts more explicit: [foreach(set) [if (condition) [expression]] However, this decrease in expressiveness seems small, and is balanced by an increase in the range of expressions that can be described. For example, we can easily describe and manipulate infinite sets in G, something that cannot be done in Setl. For brevities sake, we describe only a single expression in Setl and show how the analogous operations can be performed in G. A large number of more detailed examples can be found in [. placer notes .]. Given a set S which we know has a median value, we can produce the median value in Setl as follows: if exists x in S | #{y in S | y < x} = #{y in S | y > x} then print("median", x); The algorithm computes, for each value x, the subset of elements from S that are smaller than x, the subset of elements that are larger than S, and if these two sets are equal prints x. The solution in G works much the same way; using filters to compute the two streams. foreach (s) [if (#(s < *foreach) = #(s > *foreach)) [print("median", *foreach)] Goal Directed Evaluation We illustrate the technique of goal directed evaluation [. icon prhall, budd smalltalk .], by presenting a solution to the classic problem of placing eight queens on a chessboard in such a way that no queen can attack any other queen. The recursive solution is given a column number **column**; it returns a stream of streams, each of which represent a set of valid positions for the queens in columns 1 through **column**. Of course, in the first column any position is legal, and so we simply return the streams [[1], [2], [3], and so on. For subsequent streams it is simply a matter of trying each possible position and testing to see whether any queen to the left can attack the position, and when found returning the rows of the neighbors with the current position tacked on at the end. Notice that the local variable neighborrows is made necessary by the nested use of foreach, which prevents the access to the value returned by the outer foreach in the inner loop. rows := func(column, n) [local [neighborrows], if (column = 1) [foreach ([1..n]) [[*foreach]]] else [foreach(self(column-1, n)) [neighborrows := *foreach, foreach([1..n]) [if(not(attack(neighborrows, *foreach)) [(neighborrows, [*foreach])]]]] The function not() is a simple utility routine which returns an empty stream if its argument is nonempty, and a nonempty stream if its argument is empty: not := func(s) [if (s) [] else [1]] So we are left with the simple problem of determining whether a given position in column n can be attacked by queens resting in positions given by a stream of length n-1. attack := func(rows, newposition) [if (rows = newposition) [1], if (rows + (#rows..1 by -1) = [newposition]) [1], if (rows - (#rows..1 by -1) = [newposition]) [1]] Relational Database and Logic Programming The relational database paradigm is given natural expression in G by means of the facilities of pattern-matching expressions and output variables, and goal-directed evaluation of conditional expressions. These facilities allow relational database programming in the style that has come to be called logic programming [. mellish, shapiro .]. In sections 2.1 and 2.3 we described how databases can be described as streams of streams (streams of tuples representing records), and how the G pattern matching facility can be used to search this database. By introducing a new operator ‘‘and’’ which, like a relational, examines all pairs of its arguments and generates, for each pair, the rightmost argument, we can create more complex relations. For example, if parent is the stream given in section 2.1, we can discover the grandparent of "mary" using the following expression: if (parent[?x, "mary"] and parent[?y, x]) [print [y]] For the conjunct to be satisfied, we must first find an entry in parent which matches "mary" in the second field. Then for the second argument to the and we must find an entry which matches this value in the second component. The demand driven nature of streams in G allows us to create virtual relations in much the same fashion that prolog rules introduce relations. For example, we could encode our knowledge of grandparentship in the following construct: grandparent := [foreach(parent[?grand, ?par] and parent[par, ?child]) [(grand, child)]] Even though one may choose to think of grandparent as a function, it is not necessary to use the **func** keyword since no arguments are necessary. The grandparent stream will, on demand, consider all parents and produce the list of all grandparents. Thus we could rewrite our inquiry concerning the grandparents of mary as follows: if (grandparent[?x,"mary"]) [print [x]] At first glance one is tempted to criticize this solution on the ground that the grandparent tuple will enumerate *all* grandparent relationships, from which we are filtering out only the particular one we are interested in. Remember, however, that all results are produced in a strictly demand driven manner. Thus we only produce values as long as they are necessary for the computation. Object Oriented Programming and Abstract Datatypes A function invocation creates a new name scope, and this scope may continue to exist even after the function has returned. By means of functions returning fuctions as results, we can preserve this name space and encapsulate it in a variable. By invoking the returned function with various arguments,

we can simulate sending messages to the object which will cause it to change its local state. For example, the following definition creates an object to manage a bank account balance. Each instance of this class of objects will have one local variable, called balance. BankAccount := func(balance:0) [func(message) [[['print', [write balance]], ['add', [balance := balance + args!2]], ['subtrace', [balance := balance - args!2]]] [message, ?defn] and defn]] We can create multiple instances of bank account managers by separate invocations of the BankAccount function. We can provide an initial value, or zero will be used by default. Note that the @ operator is necessary, since we want the first (and only) value returned by the function, and not the function itself. johnsAccount := @ BankAccount(100); timsAccount := @ BankAccount(); Each of these objects maintains its own variable named balance. We can operate on these values by sending messages, using function invocation. This is the only way that these values can be accessed or changed; they are not available outside the captured scope. johnsAccount('subtract',20) timsAccount('add',10) johnsAccount('print') 90 timsAccount('print') 10 The point is not that this is necessarily a good way to program, or that the syntax is particularly simple, but that the mechanisms we have provided, pattern matching and functions as first class objects, are sufficiently powerful and combine naturally enough to permit the language to be used in a number of different ways. We are at present designing a facility for G to permit user defined types. Although all values are streams, such a type would permit the user to give names to fields in the stream, in much the same fashion as the function facility gives names to arguments. In addition, a user could define actions specific to a particular type, in effect creating abstract datatypes. Combining user defined types, deferred action lookup based on the type of the first argument, and a facility for inheritance of actions to be associated with messages from one type from another, we will end with a system which permits the creation of true abstract datatypes and the ability to program in an object oriented fashion. Conclusions The motivation that led to the design of the language G was an attempt to find an underlying data structure and a model of computation that was simple enough to result in a concise and highly expressive language, yet powerful enough to integrate many different current programming language paradigms into one cohesive and semantically consistent whole. The net result we desired was the creation of a truly multiparadigm language. It is still much too early to evaluate our success or failure at achieving this goal. The language G is an expression-based language whose basic data structure is the stream. There are no statements in the traditional sense in G, instead every expression computes its values upon demand. It is only when results are called for that code forms may be executed in G, then then only as far as necessary to satisfy an immediate request. The facilities of G; the stream datatype and the lazy evaluation semantics which permit a generate and test mode of operation, allow many of the current language paradigms to be concisely and naturally expressed in G. These paradigms can be viewed as logical extensions of the powerful underlying stream paradigm of G. Although this paper makes it sound as if the design of G is finalized, in truth this is far from being the case. We are still toying with details of the semantics of the various language features, and have not yet even started to think about implementation issues. Acknowledgements Even a cursory examination of this paper will reveal the fact that G owes a great deal and has copied features from many other programming languages. Notable among these are Icon, Seque, ABC, FP, Setl, Smalltalk and Prolog; and it goes without saying that G would not exist had not these languages come before.