DUEL — A Very High-Level Debugging Language[1]

Michael Golan[2]
David R. Hanson

Research Report CS-TR-399-92
November 1992

Abstract

Most source-level debuggers accept expressions in the source language, e.g., C, and can print source-language values. This approach is usually justified on grounds that programmers need to know only one language. But the evaluation of source-language expressions or even statements is poorly suited for making non-trivial queries about the program state, e.g., "which elements of array `x[100]` are positive?" Duel departs from the conventional wisdom: It is a very high-level language designed specifically for source-level debugging of C programs. Duel expressions are a superset of C's and include "generators," which are expressions that can produce zero or more values and are inspired by Icon, APL, and LISP. For example, `x[..100] >? 0` displays the positive elements of `x` and their indices. Duel is implemented on top of `gdb` and adds one new command to evaluate Duel expressions and display their results. This paper describes Duel's semantics and syntax, gives examples of its use, and outlines its implementation. Duel is freely available and could be interfaced to other debuggers.

[1]To appear in the Proceedings of the Winter USENIX Technical Conference, San Diego, Jan. 1993.
[2]Supported in part by NSF Grant CCR92-00790.

Introduction

Interactive source-level debuggers are now a standard part of nearly every programming environment. Most provide a rich suite of debugging facilities such as breakpoints, conditional breakpoints, watchpoints, stack traversals, etc., and many provide graphical user interfaces (GUIs) that use mouse actions and menus to invoke these facilities.

Despite these advances, the basics of debugging have changed little [8]. The basic debugging methodology is still to set breakpoints, run the program until a breakpoint is reached, and explore the program's state by displaying values of variables and data structures. GUIs make this exploration less tedious and more productive, but most are just a veneer over commands that print the values of expressions.

Programmers interact with most source-level debuggers in the language of the program being debugged, e.g., C debuggers accept and evaluate C expressions [9]. This choice is invariably justified on grounds that the programmer needs to know only one language, and that even small deviations would make a debugger unnecessarily hard to use [1, 8].

This paper investigates the contrary view: Programmers are best served by debugging languages that are more expressive and flexible than — and possibly different from — the program's source language. A concrete realization of this view is Duel, a very high-level language for debugging. Others have designed new debugger languages based, in part, on similar premises [7, 11, 12], and some recent work has focussed on semantic issues [3].

The overall "goal" of debugging is to search the program state for inconsistencies that manifest themselves as bugs. For example, questions like "which elements of array `x` are greater than `1`?," "how many nodes are in `tree`?," and "does list `L` contain two identical elements in its `value` fields?" typify the kinds of questions that can arise during state exploration.

Most debuggers can only print the values of expressions, which is of little help in answering complex queries. Some debuggers accept source-language statements or even procedures, but expressing these kinds of questions in languages such as C is tedious at best. For example, answering the query "does list `L` contain two identical elements in its `value` fields?" in C requires non-trivial code:

```
List *p, *q;
for (p = L; p; p = p->next)
   for (q = p; q; q = q->next)
      if (p->value == q->value)
         printf("%x %x contain %d\n",
            p, q, p->value);
```

This code also illustrates additional complexities, e.g., managing "debugger variables" (`p` and `q`). Also, `printf` is a poor way to display the offending values, so the debugger must provide mechanisms for accessing its display functions. Even accessing these functions with special `printf` format codes forces programmers to use non-standard facilities when debugging.

Typical debugging queries are complex enough that experienced programmers write functions whose only use is to be called from the debugger. While undoubtedly useful, this methodology is invariably inadequate because programmers cannot anticipate all of the state exploration functions that might be needed.

Duel allows many state exploration queries to be expressed concisely, often as "one-liners" without additional variables or control constructs. Other capabilities include concise ways of printing parts of large data structures. Duel is derived mostly from C, Icon [6], a very high-level string-processing language, and, to a lesser extent, from APL and LISP. Duel is implemented on top of `gdb` [13], a traditional source-level debugger for C.

Design

Duel is an expression-oriented language in which expressions can return a sequence of values. Operators permit these sequences to be manipulated in novel ways to achieve the goal of concise state exploration. As a simple example, `x[0..9] >? 1` yields the elements of `x` that are greater than `1`, and `(x,y).a` yields the `a` field of `x` and of `y`. In the first example, the "`..`" operator produces the integers `0`, `1`,..., `9`. The C indexing operator is applied to `x` and each of those integers, producing the 0th through the 9th elements of array `x`. The "`>?`" operator compares its operands like C's "`>`" operator and returns the left one when the comparison is true. Each of `x`'s values is compared with `1`, and those greater than `1` are printed along with their indices.

Duel's semantics are modeled after Icon's; Duel's syntax, however, is quite different and is described below. Icon supports *generators* — expressions that can produce zero or more values — and goal-directed evaluation, which seeks the first "successful" result by trying all possible combinations of values generated by each subexpression. In contrast, Duel has no goal-directed evaluation; it produces all of the values of its generators, except for a few special operators. In many cases, expression evaluation in Icon and Duel is similar to evaluation in other languages, e.g., `x+y` adds `x` and `y`; there is only one possible value for each operand. The semantics and efficient implementation of generators are well documented [2, 4, 5].

Icon is only one of several languages that might be used as a basis for a high-level debugging language. The use of generators in Duel is more limited than in Icon, which is a complete, general-purpose, very high-level language. In addition to its generators, Duel includes some APL-style reduction operators and operators that expand data structure alá LISP.

Duel is designed primarily to debug C, but source-language expressions in most imperative languages could be extended with generators. Most of Duel operators could apply equally well to, e.g., Pascal, PL/I, FORTRAN, or C++.

Duel's semantics are more important than its syntax. Duel is used most effectively if its semantics are well understood, but the following two sections can be read in either order. Once the basics of generators are mastered, many of their uses become idiomatic.

Semantics

Duel's semantics are best described operationally using a C-like pseudo-language that mirrors the actual implementation. This pseudo-language omits punctuation, declarations, and error checking in the obvious ways. Duel users never write in this language; it is just a descriptive convenience. A similar approach has been used to describe Icon's semantics [10].

Duel evaluates an expression by traversing its *abstract syntax* tree (AST) recursively. All AST nodes have an `op` field, which identifies the node's operand, and a `kids` field, which is an a array of pointers to the operand nodes. Nodes for specific operators have additional fields, e.g., a node for a constant includes a `constant` field that holds the constant itself. The advantage of this notation is that it is independent of a specific concrete syntax. ASTs can be specified by a simple LISP-like notation, e.g., the AST for the expression `a*5 + *b` might be

```
(plus
   (multiply (name "a") (constant 5))
   (indirect (name "b"))
)
```

If all expressions returned only one value, `eval` would be a standard tree traversal:

```
Value eval(Node n) {
   Value u, v
   switch (n->op) {
   case CONSTANT:
      return n->constant
   case NAME:
      return fetch(n->name)
```

```
   case NEGATE, INDIRECT, ...:
      u = eval(n->kids[0])
      return apply(n->op, u)
   case PLUS, MINUS, MULTIPLY, ...:
      u = eval(n->kids[0])
      v = eval(n->kids[1])
      return apply(n->op, u, v)
   ...
   }
}
```

`eval` switches on the operator, recursively evaluates the operands, if necessary, and calls `apply` to evaluate a specific operator. `fetch` retrieves the value of the variable named in the `NAME` node's `name` field. `Value` denotes a type that encapsulates all values.

Duel expressions can produce more than one value, e.g., `(1..3)+(5,9)` prints `6 10 7 11 8 12`. `(1..3)` produces `1`, `2`, and `3`, and `(5,9)` produces each of its "alternatives," `5` and `9`. The "`+`" sums all possible combinations of these values.

This feature complicates only `eval`, not the code for each individual operator, i.e., instead of changing all of the operators to take lists of values, `eval` manages the multiple values. Each call to `eval` produces one of the values. To implement this version of `eval`, state information is added to each node, and a distinguished value, `NOVALUE`, signals the end of a sequence of values. The `state` field of a node is a non-negative integer that indicates the progress of the evaluation of that node. `state` begins at `0` and is changed to `1` before the first value is returned to indicate that subsequent calls to `eval` for this node will return additional values. `goto` statements are used in the code below to emphasize this flow of control. After `NOVALUE` is returned, the next call to `eval` re-evaluates the node. The field `value` is a temporary value that must be saved between successive calls to `eval`. For example, this version of `eval` handles constants and most of the binary operators as follows; the line numbers are for explanatory purposes and are not part of the code.

```
Value eval(Node n) {
   switch (n->op) {
   case CONSTANT:
      if (n->state == 0) {
         n->state = 1
         return n->constant
      } else {
         n->state = 0
         return NOVALUE
      }
   ...
1  case PLUS, MINUS, MULTIPLY, ...:
2        if (n->state == 1) goto bin1
3  bin0: n->state = 0
4        n->value = eval(n->kids[0])
5        if (n->value == NOVALUE)
6           return NOVALUE
7        n->state = 1
8  bin1: u = eval(n->kids[1])
9        if (u == NOVALUE) goto bin0
10       v = apply(n->op, n->value, u)
11       return v
   ...
```

```
   }
}
```

To understand this code, consider the evaluation of the addition in the expression `(1..3)+(5,9)`, which has the AST

```
(plus (to 1 3) (alternate 5 9))
```

When `eval` is called with the `plus` node, control lands at line 4 above and `eval` is called with the `(to 1 3)` node. This recursive invocation of `eval` returns `1`, which is saved in the `plus` node's `value` field. The `plus` node's `state` is reset to `1`, and control ultimately lands at line 8. This second call to `eval` on `(alternate 5 9)` returns `5`, `apply` computes the sum, `6`, which is the return value from the top-level `eval`.

Duel's top-level evaluation command "drives" its expression argument and prints *all* of its values. So, `eval` is called again with the `plus` node as its argument. This time, the `plus` node's `state` is `1`, so control lands at line 8, and `eval` is called recursively for the next value from the node `(alternate 5 9)`. This call returns `9`, which causes the top-level call to `eval` to return `10`, which is printed.

Ultimately, the call to `eval` in line 8 returns `NOVALUE`, control passes to line 3, the `plus` node's `state` is reset to `0`, and line 4 calls `eval` for the next value from `(to 1 3)`. This call returns `2`, the `state` is reset to `1` again, and the whole process of re-evaluating `(alternate 5 9)` begins anew and produces `5` again.

This process continues until all of the values from `plus`'s first operand have been produced, which occurs when the call to `eval` in line 4 returns `NOVALUE`. Finally, line 6 announces that the entire `plus` expression has produced all of its values. If `eval` is called again with this `plus`, the entire evaluation process starts over because `state` has been reset to `0`.

Each of Duel's generators has a similar implementation scheme. This scheme simulates coroutines (which are similar to, but pre-date, non-preemptive threads).

Managing the `state` and `value` correctly for each generator according to its semantics is straightforward, but tedious. The semantics are conveyed equally well by assuming that `eval` is a coroutine in which the values of local variables are saved across calls, and that the statement `yield` e returns e and preserves enough information for the computation to resume after the `yield` statement. `(alternate` e_1 e_2`)` produces all of the values of e_1 followed by the values of e_2. Its detailed implementation is

```
case ALTERNATE:
      if (n->state == 1) goto alt1
      u = eval(n->kids[0])
      if (u != NOVALUE)
         return u
      n->state = 1
alt1: v = eval(n->kids[1])
      if (v != NOVALUE)
          return v
      n->state = 0
      return NOVALUE
```

The simplified code is

```
case ALTERNATE:
   while ((u = eval(n->kids[0])) != NOVALUE)
      yield u
   while ((v = eval(n->kids[1])) != NOVALUE)
      yield v
   return NOVALUE
```

Declarations, explicit comparisons with `NOVALUE`, and the final "`return NOVALUE`" are omitted when the meaning is clear, e.g.,

```
case ALTERNATE:
   while (u = eval(n->kids[0]))
      yield u
   while (v = eval(n->kids[1]))
      yield v
```

Most of the unary operators are defined by

```
case NEGATE, INDIRECT, ...:
   while (u = eval(n->kids[0]))
      yield apply(n->op, u)
```

Specific Operators

The generator (`to` e_1 e_2) produces the integers from e_1 to e_2 inclusive. The semantics of `to` are defined by

```
case TO:
   while (u = eval(n->kids[0]))
      while (v = eval(n->kids[1]))
         for (i = u; i <= v; i++)
            yield i
```

As suggested by this code, `to`'s operands can be generators, e.g.,

```
(to (alternate 1 5) (alternate 5 10))
```

produces

```
1 2 3 4 5
1 2 3 4 5 6 7 8 9 10
5
5 6 7 8 9 10
```

Some operators return one or *no* value. Duel's comparisons produce their first operand if the condition is true and nothing otherwise, e.g., (`ifgt` e_1 e_2) produces e_1 only if e_1 is greater than e_2. The implementation is

```
case IFGT, IFGE, IFLE, IFLT, IFEQ, IFNE:
   while (u = eval(n->kids[0]))
      while (v = eval(n->kids[1]))
         if (w = apply(n->op, u, v))
            yield w
```

These semantics admit generators as operands, so an expression like

```
(ifgt
   (index (name "x") (to 0 99))
   (constant 0)
)
```

produces the positive elements of the array `x[100]`.

The operators that correspond to the C operators `&&` and `||` can be problematic because, with generator operands, their semantics are nonintuitive. The semantics of `andand` illustrate the problem.

```
case ANDAND:
   while (u = eval(n->kids[0]))
      if (u != 0)
         while (v = eval(n->kids[1]))
            yield v
```

e_1 `&&` e_2 produces *all* of the values of e_2 for each non-zero value produced by e_1. When e_1 and e_2 are single-value expressions, these semantics are equivalent to C's.

The operator `(if` e_1 e_2 e_3`)` evaluates e_1; for each non-zero value of e_1, it produces all of the values of e_2, and for each zero value of e_1, it produces all of the values of e_3.

```
case IF:
   while (u = eval(n->kids[0]))
      if (u != 0)
         while (v = eval(n->kids[1]))
            yield v
      else
         while (v = eval(n->kids[2]))
            yield v
```

A sequence of expressions, `(sequence` e_1 e_2`)` evaluates e_1 but discards its values, and then produces the values of e_2.

```
case SEQUENCE:
   while (u = eval(n->kids[0]))
      ;
   while (v = eval(n->kids[1]))
      yield v
```

`(imply` e_1 e_2`)` is similar, but produces e_2's values for each value of e_1.

```
case IMPLY:
   while (u = eval(n->kids[0]))
      while (v = eval(n->kids[1]))
         yield v
```

Finally, `(while` e_1 e_2`)` produces e_2 only if *all* of the values of e_1 are non-zero:

```
case WHILE:
   for (;;) {
      while (u = eval(n->kids[0]))
         if (u == 0)
            return NOVALUE
      while (v = eval(n->kids[1]))
         yield v
   }
```

These semantics are equivalent to C's when e_1 is a single-value expression. Notice that once e_2 has produced all of its values, `while` starts over again. For example,

```
(while (index (name "x") (to 0 99)) ...)
```

produces "`...`" as long as all of the elements of `x` are non-zero.

Some operators manipulate value sequences instead of values themselves. For example, `(select` e_1 e_2`)` produces the elements of e_2 given by the integers in e_1. The implementation is described by

```
case SELECT:
   while (v = eval(n->kids[0])) {
      n->kids[1]->state = 0
      for (i = 0; i < v; i++)
         u = eval(n->kids[1])
      yield u
   }
```

Notice that e_2's `state` is reset so that it starts anew for each value of e_1. The actual implementation of `select` avoids the re-evaluation of e_2 when possible.

Several "reduction" operators reduce a sequence of values to one value, e.g., `(count` e`)` returns the number of values produced by e, `(sum` e`)` sums the values produced by e and `(equality` e_1 e_2`)` returns `1` if the values produced by e_1 are equal to those produced by e_2 and `0` otherwise.

Duel's evaluation mechanism also applies to calls to functions in the target program. If any of the arguments are generators, the function is called repeatedly for all combinations of values, e.g.,

```
printf("%d %d, ", (3,4), 5..7)
```

prints

```
3 5, 3 6, 3 7, 4 5, 4 6, 4 7,
```

Aliases

As suggested above, `(name "`x`")` fetches the value of the variable x. x can be a variable in the target program or an *alias*. Aliases are created by `(define` a e`)`, which defines a to be an alias for e. If e is an lvalue, so is a, e.g., after `(define b x[5])`, changing `b` changes `x[5]`. If e is a generator, a is aliased to each value in turn and `define` returns those values.

```
case DEFINE:
   while (u = eval(n->kids[1]) {
      alias(n->name, u)
      yield u
   }
```

The operator `(with` e_1 e_2`)` evaluates e_2 in the "scope" of e_1. When e_1 is a structure, "opening the scope" of e_1 makes the fields visible as ordinary identifiers. Names in e_2 refer to the appropriate fields in e_1; for example, if `x` and `y` are instances of structures with a field `f`,

```
(with
   (alternate (name "x") (name "y"))
   (alternate (name "f") (name "g"))
)
```

generates `x.f`, `x.g`, `y.f`, and `y.g`. The semantics of `with` are defined as follows.

```
case WITH:
   while (u = eval(n->kids[0]) {
      push(u)
      while (v = eval(n->kids[1])
         yield v
      pop()
   }
```

The `push` and `pop` functions manipulate the name-resolution stack used by `fetch`. Also, the special name "`_`" in e_2 refers to the value of e_1.

The `with` operator is used by other operators to traverse data structures. `(dfs` e_1 e_2`)` "expands" the data structure e_1 using e_2 to indicate the traversal path as follows. Unvisited nodes are kept in a stack. At each step, the top of the stack, X, is popped, the non-null values generated by `(with *`X e_2`)` are pushed onto the stack, and `dfs` yields X. This process continues until the stack is empty. In the semantics below, `stack` and `unstack` manipulate `n`'s traversal stack and `push` and `pop` manipulate the name-resolution stack described above.

```
case DFS:
   while (u = eval(n->kids[0]) {
      stack(n, u)
      while (v = unstack(n)) {
         push(v)
         while (w = eval(n->kids[1])
            stack(n, w)
         pop()
         yield v
      }
   }
```

If `head` is a pointer to a linked list in which nodes are linked via `next` fields,

```
(dfs (name "head") (name "next"))
```

generates the elements of the list. Likewise,

```
(dfs
   (name "root")
   (alternate (name "left") (name "right"))
)
```

generates the nodes in a binary tree in preorder. (The actual implementation stacks the values of e_2 in reverse order so that the nodes are visited in the expected order.) Other operators do similar expansions with different orderings, e.g., breadth-first search.

Syntax

Duel uses an extended C-like concrete syntax to specify the semantics described above. Duel accepts expressions, compiles them into ASTs, evaluates them, and prints the resulting values. Expressions include all of the C operators with the expected semantics except for "`,`", and C's scope rules apply. Control structures, like `for`, `if`, etc. are cast as expressions, not statements, much as in Icon. Finally, there are numerous Duel-specific operators that specify the generators described above.

In the absence of generators, Duel expressions are essentially equivalent to a debugger's "print" command, e.g.,

```
gdb> print 1 + (double)3/2
2.500
gdb> duel 1 + (double)3/2
2.500
```

As this example suggests, Duel is an extended version of `gdb`; the `duel` command is similar to `gdb`'s `print` command, except that the `duel` command drives its expression argument and prints all of its values, e.g.,

```
gdb> duel (1,2,5)*4+(10,200)
14 204 18 208 30 220
gdb> duel (3,11)+(5..7)
8 9 10 16 17 18
```

The comma operator is the concrete syntax for the `alternate` operator described in the previous section; e_1`,`e_2 produces all of e_1's values followed by e_2's values. The operator "`..`" specifies the `to` operator; e_1`..`e_2 produces the integers from e_1 to e_2 inclusive.

Duel treats lvalues and rvalues as in C. For example, suppose that `hash` is defined by the declaration

```
struct symbol {
   char *name;
   int scope;
   struct symbol *next;
} *hash[1024];
```

which is a typical representation for symbol tables in compilers. `hash` is an array of pointers to lists of `symbol` structures, the lists are threaded through the `next` fields, and the `symbol`s are in decreasing order of the `scope` fields. The command

```
gdb> duel hash[0..1023]->scope = 0 ;
```

clears the `scope` field of the first `symbol` on each list. `hash[0..1023]` produces lvalues; the semantics of C's assignment are unchanged. This example produces no output; the terminating semicolon causes the expression to be evaluated for side effects only.

The operator `>?` specifies the operator `ifgt`; e_1 `>?` e_2 returns e_1 if e_1 is greater than e_2 and nothing otherwise. This operator and the similar ones for the other comparisons can be used with other generators like "`..`" to search for specific values. For example,

```
gdb> duel x[1..4,8,12..50] >? 5 <? 10
x[3] = 7
x[18] = 9
x[47] = 6
```

searches portions of `x` for values that are between `5` and `10`. Duel's output includes symbolic expressions that suggest the derivation of the values printed. Thus, the output from the search shows not only the desired values, but also pinpoints the elements of `x` that hold those values. (The examples at the beginning of this section omitted the symbolic output.) The command `x[1..4,8,12..50] ==? (6..9)` is another formulation of the same search.

Duel also supports the C operators, `==`, etc., but their semantics are as in C, e.g.,

```
gdb> duel x[1..3] == 7
x[1]==7 = 0
x[2]==7 = 0
x[3]==7 = 1
```

prints *all* of the indices and values of `x`.

The unary expression "`..`e" is shorthand for `0..`e`-1` and is useful for indexing arrays. For example,

```
gdb> duel (hash[..1024] !=? 0)->scope >? 5
hash[42]->scope = 7
hash[529]->scope = 8
```

prints the elements in `hash` that have a `scope` value greater than `5`.

This example illustrates the crux of the problem in designing Duel's syntax. It must necessarily be a superset of C, but the wealth of operators quickly overwhelms the vocabulary that permits a readable notation for them.

In programming language design, readability is important because programs are read more than written, e.g., in debugging and maintenance. Duel expressions, however, are ephemeral; they exist only long enough to be executed once. They are written once and read at most once. Duel's syntax is designed to facilitate, on-the-fly, left-to-right composition, e.g., `hash[..1024]` specifies all of the lists, `!=? 0` specifies those that are non-null, `->scope` specifies the `scope` fields of just those elements, and `>? 5` limits the output to the desired elements. While these kinds of expressions appear cryptic initially, they become idiomatic with use. At the very least, Duel expressions are more compact than the equivalent C code, e.g., the C (and Duel) code for the search just described is

```
gdb> duel int i;
   for (i = 0; i < 1024; i++)
      if (hash[i] != 0)
         if (hash[i]->scope > 5)
            printf("hash[%d]->scope = %d\n",
               i, hash[i]->scope);
```

Duel declarations, e.g., `int i`, establishes aliases to newly allocated target locations.

Duel accepts most of C, and C and Duel expressions can be intermixed freely. For example, the following Duel lines all print the same `scope` fields as the search of `hash` described above.

```
gdb> duel int i; for (i = 0; i < 1024; i++)
      if (hash[i] && hash[i]->scope > 5)
         hash[i]->scope

gdb> duel int i; for (i = 0; i < 1024; i++)
      if (hash[i]) hash[i]->scope >? 5

gdb> duel int i; for (i = 0; i < 1024; i++)
      (hash[i] !=? 0)->scope >? 5
```

As suggested by its semantics, Duel's `if` is an expression, e.g.,

```
gdb> duel for (i = 0; i < 9; i++)
             4 + if (i%3==0) i*5
4+i*5 = 4
4+i*5 = 19
4+i*5 = 34
```

The appearance of "`i`" instead of its value in this example illustrates a potentially unappealing side effect of Duel's symbolic display algorithm. The algorithm substitutes the actual value only for generators; other expressions are displayed as entered. Enclosing an expression in braces overrides the default display for that expression and causes its value to be displayed, e.g.,

```
gdb> duel for (i = 0; i < 9; i++)
             4 + if (i%3 == 0) {i}*5
4+0*5 = 4
4+3*5 = 19
4+6*5 = 34
```

The semicolon specifies the `sequence` operator, which evaluates but discards its left operand, and returns its right operand, e.g.,

```
gdb> duel i := 1..3; i + 4
i+4 = 7
```

`imply` is specified by `=>`; e_1`=>`e_2 produces e_2's values for each of e_1's values, e.g.,

```
gdb> duel i := 1..3 => {i} + 4
1+4 = 5
2+4 = 6
3+4 = 7
```

The operator a `:=` e defines a to be an alias for e, which may be either an lvalue or an rvalue, e.g.,

```
x := hash[..1024] !=? 0 =>
   y := x->scope => y = 0
```

clears the `scope` fields of the `symbols` in `hash`. `x` is an alias for each element of `hash` and `y` is an alias for each `scope` field.

The operators "`.`" and `->` specify Duel's `with` operator; as in C, "`.`" applies to structures and `->` applies to pointers to structures. In both e_1`.`e_2 and e_1`->`e_2, e_2 is evaluated within the scope of e_1. For example, alternation can specify several fields of a structure:

```
gdb> duel hash[1,9]->(scope,name)
hash[1]->scope = 3
hash[1]->name = "x"
hash[9]->scope = 2
hash[9]->name = "abc"
```

The "`.`" and `->` operators are quite general, e.g.,

```
x := hash[..1024] !=? 0 =>
   x->(if (scope > 5) name)
```

prints the `name` field of the elements in `hash` that have a `scope` greater than `5`. References to "`_`" refer to `with`'s operand, which helps eliminate temporaries like `x` in the example above; for instance, the example above can be done by

```
hash[..1024]->(if (_ && scope > 5) name)
```

Using "`_`" instead of an alias often produces more informative output. For example

```
gdb> duel y := x[..10] =>
            if (y < 0 || y > 100) y
y = -9
y = 120

gdb> duel x[..10].if (_ < 0 || _ > 100) _
x[3] = -9
x[8] = 120
```

The first command uses an alias for each element of `x` and prints those elements that are less than `0` or greater than `100`. The output displays the name of the alias, not the elements of `x`. The "`_`" stands for the value itself, an element of `x` in this example, so the output of the second command displays the specific elements of `x` that are generated. The same effect can be achieved with aliases but requires another temporary:

```
y := x[j := ..10] =>
   if (y < 0 || y > 100) x[{j}]
```

The operator `-->` specifies the `dfs` node described in the previous section. e_1`-->`e_2 produces the values from the data structure given by e_1 using e_2 to specify the traversal. For example, if `head` points to a linked list of structures threaded through `next` fields, `head-->next` produces the elements of the list, i.e., it produces `head`, `head->next`, `head->next->next`, etc., until a `NULL` pointer or an invalid pointer terminates the sequence. So,

```
gdb> duel hash[0]-->next->scope
hash[0]->scope = 4
hash[0]->next->scope = 3
hash[0]->next->next->scope = 2
hash[0]->next->next->next->scope = 1
```

prints the `scope` fields of the list emanating from `hash[0]`. Specifying `hash[..1024]` would print the `scope` fields for all of the `symbols` in the table.

The expression

```
L-->next->(value ==? next-->next->value)
```

answers the Introduction's query about list `L` containing identical elements in its `value` fields. `L-->next` generates each element in `L`, the `value` field of which is compared to the `value` fields of each of the succeeding elements generated by `next-->next`. Compare this compact expression with the C code given in the Introduction. The longer C code hides a bug: the initialization of the inner `for` loop should be `q = p->next`.

Suppose a binary tree is composed of nodes in which each node includes an integer `key` and `left` and `right` fields that point to the subtrees, and that `root` is the head of the binary tree specified in preorder as `(9 (3 (4) (5)) (12))`. The keys in the entire tree are printed by

```
gdb> duel root-->(left,right)->key
root->key = 9
root->left->key = 3
root->left->left->key = 4
root->left->right->key = 5
root->right->key = 12
```

and the path to the node holding `5` is printed by

```
gdb> duel root-->(if (key < 5) left
             else if (key > 5) right)->key
root->key = 9
root->left->key = 3
root->left->right->key = 5
```

Another, more complex, example is

```
gdb> duel hash[..1024]-->next->
             if (next) scope <? next->scope
hash[287]-->next[[8]]->scope = 5
```

`hash[..1024]-->next` produces all of the nodes on all of the lists in `hash`. The `if` expression returns a `scope` field only if it is less that the `scope` field of the next element on the list. Thus, this command verifies that the `symbols` in each list are sorted in decreasing order of `scope`, as expected. This output displays the error. The symbolic display algorithm automatically prints occurrences of `->`a`->`a as `-->`a`[[2]]`, etc.

The `select` operator is specified by e_1`[[`e_2`]]` and produces the values from e_1 as specified by the values in e_2. For example,

```
gdb> duel ((1..9)*(1..9))[[52,74]]
6*8 = 48
9*3 = 27
gdb> duel hash[287]-->next[[7..10]]->scope
hash[287]-->next[[7]]->scope = 6
hash[287]-->next[[8]]->scope = 5
hash[287]-->next[[9]]->scope = 9
hash[287]-->next[[10]]->scope = 3
```

The reduction operators help summarize the contents of data structures, e.g., `count` is specified by `#/`*e* and counts the number of values produced by *e*:

```
gdb> duel #/(root-->(left,right)->key)
5
```

The operator unary *e*`#`*n* produces the values of *e* and arranges for *n* to be an alias for the index of each value in *e*. Thus, if `L` is the list mentioned in the Introduction and its 4th and 9th nodes each contain `27` the following command displays the duplication.

```
gdb> duel L-->next#i->value ==?
          L-->next#j->value =>
             if (i < j)
                L-->next[[i,j]]->value
L-->next[[4]]->value = 27
L-->next[[9]]->value = 27
```

The expression *e*`@`*n* produces the values of *e* until *e*`.`*n* is non-zero. For example, is `s` is a pointer to a character, `s[0..999]@(_=='\0')` produces `s[0]`, `s[1]`, ...up to but not including the terminating null character. Also, *n* can be a constant, in which case the expression produces the values of *e* up to the first one that equals *n*. "*e*`..`" generates an essentially infinite sequence of integers beginning at *e*, so `argv[0..]@0` generates the strings in `argv`.

Implementation

Duel is designed to be implemented as an add-on to existing debuggers. Currently, it is interfaced only to `gdb`, but Duel is not derived in any way from `gdb`. Duel works wherever `gdb` does and can be used with `emacs` and other debugger front ends.

Duel's `yacc`-based parser and the hand-written lexer accept a Duel expression and compile it into an abstract syntax tree. The nodes in the AST correspond to the primitive operators described above.

Evaluation is implemented by `duel_eval`, which is the actual function corresponding to the abstract function `eval` use to describe the semantics. `duel_eval`'s code for most of the operators is equivalent to the pseudo-code that describes their semantics.

`duel_eval` and its associated functions are about 400 lines of C. Related functions, which manipulate search stacks, aliases, etc., are another 300 lines, and the operator application functions, including `Value` manipulations, consist of about another 1200 lines. The graph-expansion operators, e.g., `-->`, are implemented as described above, but the current implementation does not handle cycles.

As for other very high-level languages, type checking must be done during evaluation, not during compilation. For example, in `(x,y).a`, `x` and `y` can each have any structure type with a field named `a`. Consequently, the ASTs are decorated with symbolic values, like `a`, instead of pointers to symbol-table entries as in most compilers.

While evaluation-time type checking is flexible, it costs time. For example, most of the time in evaluating `1..100+i` goes to the `100` lookups of `i`. The current implementation of `duel_eval` is flexible to allow experiments with different semantics and syntax, but more efficient implementations of generators are possible [14].

The evaluation time for most Duel expressions is negligible. For example, `x[..10000] >? 0` compiles and executes in about 5 seconds on a DECStation 5000. A faster implementation would be required if Duel expressions were used in watchpoints and conditional breakpoints. For many Duel expressions, run-time type checking and symbol lookup could be done at compile time using type-inference techniques.

The "values" produced during evaluation have a type, an actual value, and a symbolic value. The actual value is a value of a primitive C type or an lvalue, which is a pointer to target data. The symbolic value is a symbolic expression (i.e., a legal Duel expression) that indicates how the value was computed. The symbolic value of a variable is its name; for most binary operators $\times$, the symbolic value is $a \times b$ where a and b are the symbolic values of the operands. Some operators have symbolic values that relate better to the computation at hand, e.g., `a..b`'s symbolic value is the current iteration value. Symbolic values assist in the display of results as well as errors: The offending operand's symbolic value is printed, e.g., the expression `ptr[..99]->val` might produce

```
Illegal memory reference in x of x->y:
   ptr[48] = lvalue 0x16820.
```

The symbolic value of an expression is computed at the same time that the expression is evaluated, e.g., in `x[1+2]` the strings `"1+2"` and `"x"` are combined to produce `"x[1+2]"` at the same time that the lvalue `&x+3` is computed. In most cases, the computation of the symbolic value is more expensive than computing the result. Furthermore, many of the symbolic computations are unnecessary, because they are never printed, e.g., in `x[..1000] !=? 0`, the symbolic expression `x[`*i*`]` is computed 1000 times, even though it might be printed only once. This kind of overhead is noticeable in complex queries and would need to be eliminated if such queries were used in watchpoints and conditional breakpoints.

Duel's interface to a debugger is a two-way interface and is intentionally narrow to simplify connecting it to a debugger. Duel duplicates some debugger capabilities in order to reduce its dependence on specific debuggers. For example, Duel contains its own type and value representations and its own implementation of the C operators.

The only new **gdb** command, `duel` *expr*, accepts a Duel expression and passes it as a string to Duel's single entry point. The only modification to **gdb** was the change to one line to allow `#` in commands (`#` starts a comment in **gdb**; Duel uses `##`). A single module contains the interface code between `Duel` and **gdb**. This module is about 400 lines of C broken down as follows.

30	`duel` command
100	converting between **gdb** and Duel types
100	symbol-table functions
70	accessing the target's address space
100	miscellaneous

Duel calls functions to allocate memory, read and write the target's data space, and to determine the types and addresses of target symbols. It does not call any **gdb** functions.

Duel's debugger interface consists of the following functions.

duel_get_target_bytes
duel_put_target_bytes:
copies **n** bytes to/from a target address.

duel_alloc_target_space:
allocates **n** bytes in the target space.

duel_call_target_func:
calls a function in the target.

duel_get_target_variable:
returns value/type information for a symbol.

duel_get_target_typedef/struct/union/enum:
returns type information for a symbol.

Except for type and value conversions, most of these functions simply call `gdb` equivalents. Only a few other miscellaneous functions are needed, e.g., to find the number of active frames, to retrieve bit fields in a machine-dependent way, etc.

Duel has been "ported" only from `gdb` 4.2 to `gdb` 4.6 on both SUN and DEC workstations. It has also been tested as a stand-alone program under MS-DOS. The change in `gdb` versions required modifications to only 4 lines of code in the interface module because internal `gdb` structures changed.

Discussion

Initial experience with Duel suggests that its generators are an effective way to explore program state. Once the initial implementation was working, it was used to probe both itself and `gdb`. This exploration not only uncovered bugs, but helped to understand the inner workings of `gdb`, which was necessary for designing and implementing the Duel–debugger interface.

As expected, Duel's syntax remains a potential hurdle. Understanding the semantics independently of the syntax helps, but programmers must interact with the debugger at some syntactic level, so Duel's syntax continues to evolve. Alternatives are also under consideration. For example, some database query languages use a visual programming approach to composing queries. Duel might benefit from a similar approach, especially if it maintained a history so that common, program-specific queries could be made by simply pointing and clicking. Allowing such history lists to be edited might also help.

Currently, Duel expressions can refer only to program variables. For example, displaying the local `x` in *all* of the currently active stack frames for the function that declares `x` is tedious to do with most debuggers. Mechanisms for exploring such "unnamed" portions of the program state would be useful and are under investigation. Duel would also be useful in other traditional debugging facilities, e.g., watchpoints and conditional breakpoints.

Duel's linguistic framework might apply to other programming environment facilities that rely on program state exploration. Assertions, for example, make claims about the state at various points in a program. Complex assertions, e.g., "`x[0]` through `x[n]` are positive," often need non-trivial code to compute the assertion outcome. Annotating programs with assertions written in a Duel-like language might simplify making these kinds of assertions and encourage their use.

Availability

Duel is public-domain software. It is available for anonymous `ftp` from `ftp.cs.princeton.edu` in the directory `pub/duel`.

References

[1] B. Beander. `VAX DEBUG`: An interactive, symbolic, multilingual debugger. *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, 18(8):173–179, Aug. 1983.

[2] T. A. Budd. An implementation of generators in C. *Journal of Computer Languages*, 7(2):69–82, Mar. 1982.

[3] R. H. Crawford, R. A. Olsson, W. W. Ho, and C. E. Wee. Semantic issues in the design of languages for debugging. In *Proceedings of the International Conference on Computer Languages*, pages 252–261, Oakland, CA, Apr. 1992.

[4] R. E. Griswold. The evaluation of expressions in Icon. *ACM Transactions on Programming Languages and Systems*, 4(4):563–584, Oct. 1982.

[5] R. E. Griswold and M. T. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, Princeton, NJ, 1986.

[6] R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[7] M. S. Johnson. The design of a high-level, language independent symbolic debugging system. In *Proceedings of the ACM Annual Conference*, pages 315–322, Seattle, WA, Oct. 1977.

[8] M. S. Johnson. *The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment.* PhD thesis, The University of British Columbia, Aug. 1978.

[9] M. A. Linton. The evolution of Dbx. In *Proceedings of the Summer USENIX Technical Conference*, pages 211–220, Anaheim, CA, June 1990.

[10] J. O'Bagy and R. E. Griswold. A recursive interpreter for the Icon programming language. *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices*, 22(7):138–149, July 1987.

[11] R. A. Olsson, R. H. Crawford, and W. W. Ho. Dalek: A GNU, improved programmable debugger. In *Proceedings of the Summer USENIX Technical Conference*, pages 221–231, Anaheim, CA, June 1990.

[12] R. A. Olsson, R. H. Crawford, W. W. Ho, and C. E. Wee. Sequential debugging at a high level of abstraction. *IEEE Software*, 8(3):27–35, May 1991.

[13] R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, 1991.

[14] K. Walker and R. E. Griswold. An optimizing compiler for the Icon programming language. *Software—Practice & Experience*, 22(8):637–657, Aug. 1992.