NAME

duel − A high level C debugging language extension to gdb

SYNOPSIS

duel [gdb options] [*prog*[*core* | *procID*]]

DESCRIPTION

Duel is a special purpose language designed for concise state exploration of debugged C programs, currently implemented under the GNU debugger *gdb*(1). Duel is invoked by entering the shell command *duel* instead of *gdb*. It is identical to gdb except for comments, which begin with ‘##’ instead of ‘#’, and the new *dl* command for Duel expressions:

```
gdb> dl x[1..10] >? 5
x[3] = 14
x[8] = 6
```

prints the array elements x[1] to x[10] that are greater than 5. The output includes the values 14 and 6, as well as their symbolic representation "x[3]" and "x[8]".

IF YOU NEVER USED GDB

The improved functionality added by Duel merits a fresh look even by debugger shunners. Gdb is a powerful debugger with many commands, a thick manual and various interfaces including *emacs*(1) and *xxgdb*(1). These gdb commands should help you get started:

b line	set a breakpoint at the line (b func to break at a function)
d n	delete breakpoint number n (gdb prints n when bp occurs)
l line	list the source be ginning at line (l file.c:line for module)
r parm	run/restart the program with the given parameters
s	single-step to the next statement (steps into function calls)
n	single-step to the next line, skipping over function calls
c	continue execution
bt	show a stack trace
p exp	evaluate a symbolic expression
dl exp	evaluate a Duel expression
dl gdb	gi ve a gdb command summary

The most common use is ‘b func’ followed by ‘r’ followed by several ‘n’ and ‘s’, evaluating expressions in between.

DUEL QUICK START

Duel is implemented by adding the *dl* command to gdb. All gdb commands work as before. The dl command, however, is interpreted by duel. Gdb concepts (such as the value history) do not work in the dl command, and duel concepts are not understood by other gdb command.

Duel is based on expressions which return multiple values. The x..y operator returns the integers from x to y; the x,y operator returns x and then y, e.g.

```
gdb> dl (1,9,12..15,22)
```

prints 1, 9, 12, 13, 14, 15 and 22. Such expressions can be used wherever a single value is used, e.g.

```
gdb> dl x[0..99]=0 ;
```

assigns zero to the first 100 elements of array x. The semantics of x[i] are the same as in C. They are applied for each of the values returned by 0..99, which can be thought of as an implied external loop. The trailing semicolon indicates evaluation for side-effects only, with no output. Duel incorporates C operators, casts C statements as expressions, and supports limited variable declaration:

```
gdb> dl int i;for(i=0;i<100;i++)
            if(x[i]<0) printf("x[%d]=%d\n",i,x[i]);
x[7] = -4
```

The semicolon prevents Duel output; only output from printf is printed. Aliases are defined with x:=y and provide an alternative to variable declaration. We could also return x[i] instead of using printf:

```
gdb> dl if(x[i:=0..99]<0) x[i]
x[i] = -4
```

The symbolic output "x[i]" can be fixed by surrounding i with {}, i.e.

```
gdb> dl if(x[i:=0..99]<0) x[{i}]
x[7] = -4
```

The {} are like (), but force the symbolic evaluation to use i's value, instead of "i". You can usually avoid this altogether with direct Duel operators:

```
gdb> dl x[..100] <? 0
x[7] = -4
```

The ..n operator is a shorthand for 0..n-1, i.e. ..100 is the same as 0..99. The x<?y, x==?y, x>=?y, etc., operators compare their left side operand to their right side operand as in C, but return the left side value if the comparison result is true. Otherwise, they look for the next values to compare, without returning anything.

Duel's x.y and x->y allow an expression y, evaluated under x's scope:

```
gdb> dl emp[..100].(if(code>400) (code,name))
emp[46].code = 682
emp[46].name = "Ela"
```

The if() expression is evaluated under the scope of each element of emp[], an array of structures. In C terms, we had to write:

```
gdb> dl int i; for(i=0; i<100 ; i++)
        if(emp[i].code>400) emp[{i}].code,emp[{i}].name
```

A useful alternative to loops is the x=>y operator. It returns y for each value of x, setting '_' to reference x's value, e.g.

```
gdb> ..100 => if(emp[_].code>400) emp[_].code,emp[_].name
```

Using '_' instead of 'i' also avoids the need for {i}. Finally, the x−−>y operator expands lists and other data structures. If head points to a linked list threaded through the next field, then:

```
gdb> dl head-->next->data
head->data = 12
head->next->data = 14
head-->next[[2]]->data = 20
head-->next[[3]]->data = 26
```

produce the data field for each node in the list. x−−>y returns x, x->y, x->y->y, x->y->y->y, ... until a

NULL is found. The symbolic output "x−−>y[[n]]" indicates that ->y was applied n times. x[[y]] is also the selection operator:

gdb> dl head-->next[[50..60]]->data

return the 50th through the 60th elements in the list. The #/x operator counts the number of values, so

gdb> dl #/(head-->next->data >? 50)

counts the number of data elements over 50 on the list. Several other operators, including x@y, x#y and active call stack access are described in the operators section.

OPERATORS SUMMARY

Assoc	Operators	Details
left	{} () [] -> . f() -->	x-->y expands x->y x->y->y ...
	x[[y]] x#y x@y	generate x; select, index or stop-at y
right	#/ - * & ! ˜ ++ -- (cast)	#/x number of x values
	frame(n) sizeof(x)	reference to call stack level n
left	x/y x*y x%y	multiply, divide, reminder
left	x-y x+y	add, subtract
left	x<<y x>>y	shift left/right
none	x..y ..y x..	..y = 0..y-1. x..y return x, x+1...y
left	< > <= >= <? >? <=? >=?	x>?y return x if x>y
left	== != ==? !=?	x==?y return x if x==y
left	x&y	bit-and
left	xˆy	bit-xor
left	x|y	bit-or
left	x&&y &&/x	&&/x are all x values non-zero?
left	x||y ||/x	||/x is any x value non-zero?
right	x? y:z	foreach x, if(x) y else z
right	x:=y x=y x+=y ...	x:=y set x as an alias to y
left	x,y	return x, then y
right	x=>y	foreach x, evaluate y with x value ‘_’
right	if() else while() for()	C statements cast as operators
left	x;y	evaluate and ignore x, return y

EXAMPLES

dl (0xff-0x12)*3	compute simple expression
dl (1..10)*(1..10)	display multiplication table
dl x[10..20,22,24,40..60]	display x[i] for the selected indexes
dl x[9..0]	display x[i] backwards
dl x[..100] >? 5	display x[i] that are greater than 5
dl x[..100] >? 5 <? 10	display x[i] if 5<x[i]<10
dl x[..100] ==? (6..9)	same
dl x[0..99]=>if(_>5 && _<10) _	same
dl y[x[..100] !=? 0]	display y[x[i]] for each non-zero x[i]
dl emp[..50].code	display emp[i].code for i=0 to 49
dl emp[..50].(code,name)	display emp[i].code & emp[i].name
dl val[..50].(is_dbl? x:y)	display val[i].x or val[i].y depending on val[i].is_dbl.
dl val[..50].if(is_dbl) x else y	same as above
dl x[..100]=0 ;	assign 0 to x[i]
dl x[i:=..100]=y[i] ;	assign y[i] to x[i]
dl x[..100]=y[..100] *ERR*	assign y[99] to each x[j]

dl x[i:=..3]=(4,5,9)[[i]]	assign x[0]=4 x[1]=5 x[2]=9
dl x[..3]=(4,5,9) *ERR*	assign 9 to each element
dl if(x[i:=..100]<0) x[i]=0 ;	assign 0 to negative x[i]
dl (hash[..1024]!=?0)->scope	hash[i].scope for non-null hash[i]
dl x[i:=..100] >? x[i+1]	check if x[i] is not sorted
dl x[i:=..100] ==? x[j:=..100]=>	checks if x has non-unique elements
if(i<j) x[{i,j}]	
dl if(x[i:=..99] ==	same
x[j:=i+1..99]) x[{i,j}]	
dl (x[..100] >? 0)[[0]]	the 1st (0th element) positive x[i]
dl (x[..100] >? 0)[[2]]	return the 3rd positive x[i]
dl (x[..100] >? 0)[[..5]]	return the first 5 positive x[i]
dl (x[0..] >? 6)[[0]]	return the first x[i]>6, no limit on i
dl argv[0..]@0	argv[0] argv[1] .. until first null
dl x[0..]@-1 >? 9	x[0..n]>9 where n is first x[n]== -1
dl emp[0..]@(code==0)	emp[0]..emp[n-1] where emp[n].code==0
dl head-->next->val	val of each element in a linked list
dl head-->next[[20]]	the 21st element of a linked list
dl *head-->next[[20]]	display above as a struct
dl strcmp(head-->next->msg,	search linked list for a string
"testing") ==? 0	
dl #/head-->next	count elements on a linked list
dl x-->y[[#/x-->y - 1]]	last element of a linked list
dl x-->y[[#/x-->y - 10..1]]	last 10 elements of a linked list
dl head-->next->	check if the list is sorted by val
if(next) val >? next->val	
dl head-->(next!=?head)	expand cyclic linked list (tail->head)
dl head-->(next!=?_)	handle termination with p->next==p
dl root-->(left,right)->key	expand binary tree, show keys
dl root-->(left,right)->(	check bin tree sorted by key
(left!=?0)->key>=?key, (right	!=?0)->key<=?key)
dl (1000..=>if(&&/(2,3.._-1=>__%_	find first 10 primes over 1000
) _)[[..10]]	
dl (T mytype) x	convert x to user defined type mytype
dl (struct s*) x	convert x to struct s pointer
dl if(x) y; else z *ERR*	';' must be followed by an expression
dl {x} y *ERR*	'}' requires ';' if followed by exp

SEMANTICS

Duel's semantics are modeled after the Icon programming language. The input consists of expressions which return sequences of values. C statements are cast as expressions, too. Expressions are parsed into abstract syntax trees, which are traversed during evaluation. The evaluation of most nodes (operators) recursively evaluates the next value for each operand, and then applies the operator to produce the next result. Only one value is produced each time, and Duel's eval function keeps a 'state' for each node (backtracking, co-routines, consumer-producer or threads are good metaphors for the evaluation mechanism.)

For example, in (5,3)+6..8, the evaluation of '+' first retrieves the operands 5 and 6, to compute and return 5+6. Then 7, the next right operand is retrieved and 5+7 is returned, followed by 5+8. Since there are no other right operand value, the next left operand, 3 is fetched. The right operand's computation is restarted returning 6, and 3+6 is returned. The final return values are 3+7 and 3+8.

The computation for operators like x>?y is similar, but when x<=y, the next values are fetched instead of returning a value, forming the basis for an implicit search. Operators like '..' return a sequence of values for each pair of operands. For a better understanding of the evaluation mechanism, see the USENIX Winter/93 conference paper "DUEL - A Very High Level Debugging Language".

Duel values follow the C semantics. A value is either an "lvalue" (can be used as the left hand side of assignment), or an "rvalue". Therefor, objects like arrays can not be directly manipulated (However, operators like x..y can accomplish such tasks.)

Duel types also follow the C semantics, with some important differences. C types are checked statically; Duel types are checked when operators are applied, e.g., (1,1.0)/2 returns 0 (int) and 0.5 (double); (x,y).z returns x.z and y.z even if x and y are of different types, as long as they both have a field z.

Values and types of symbols are looked up at run-time (using gdb's lookup rules), allowing dynamic scoping and types, but causing a parsing problem: (x)(y) can be parsed as either a function call x(y) or a cast (x)y; x*y can be parsed as a declaration of y as (x*) or as multiplication.

To avoid this ambiguity, the keyword T must precede a user defined type. For example, if value is a typedef, C's (value (*)()) x is written in Duel as: (T value (*)()) x. Types that begin with a reserved keyword don't need T, e.g. (struct value*) x and (long *[5]) y are accepted. As special cases, (type)x and (type*)x are accepted but discouraged (it causes (printf)("hi"), which is valid in C, to fail). A side effect is that "sizeof x" must be written as sizeof(x).

OPERATORS

*x+y x-y x*y x/y x%y x^y x|y x&y x<<y x>>y*
x>y x<y x>=y x<=y x==y x!=y x=y x[y]

These binary operators follow their C semantics. For each value of x, they are evaluated for every value of y, .e.g. (5,2)>(4,1) evaluates as 5>4, 5>1, 2>4, 2>1 returning 1, 1, 0, 1. The y values are re-evaluated for each new value of x, e.g. i=4; (4,5)>i++ evaluates as 4>4 and 5>5. Beware of multiple y values in assignment, e.g. x[..3]=(4,6,9) does not set x[0]=4, x[1]=6 and x[2]=9. It assigns 4, 6 and 9 to each element, having the same effect as x[..3]=9. Use x[i:=..3]=(4,6,9)[[i]] to achieve the desired effect.

*-x ~x &x *x !x ++x --x x++ x-- sizeof(x) (type)x*

These unary operators follow their C semantics. They are applied to each value of x. The increment and decrement operators require an lvalue, so i:=0 ; i++ produces an error because i is an alias to 0, an rvalue. Parenthesis must be used with sizeof(x), "sizeof x" is not allowed. Cast to user defined type requires generally requires T, e.g.,
(T val(*)())x, but (val)x and (val*)x are accepted as special cases.

x&&y x||y

These logical operators also follow their C semantics, but have non-intuitive results for multi-valued x and y, e.g. (1,0,0) || (1,0) returns 1,1,0,1,0 -- the right hand-side (1,0) is returned for each left-hand side 0. It is best to use these operators only in single value expressions.

x? y:z if(x)y if(x)y else z

These expressions return the values of y for each non-zero value returned by x, and the values of z for each zero value returned by x, e.g. if(x[..100]==0) y returns y for every x[i]==0, not if all x[i] are zero (if(&&/(x[..100]==0)) y does that). Also, "if(x) y; else z" is illegal. Duel's semicolon is an expression separator, not a terminator.

while(x)y for(w;x;y)z

The while(x)y expression returns y as long as all values of x are non-zero. The for() expression is similar and both have the expected C semantics. For example, "for(i=0 ; i<100 ; i++) x[i]" is the same as x[..100]. Unlike the if() expression, while(x[..100]==0) continue to execute only if all elements of x are zero, i.e. the condition is evaluated into a single value using an implicit &&/x.

Variable declaration: *type name [,name ...] ; ...*

Expressions can begin with variables declaration (but not initialization). Internally, a declaration sets an alias to space allocated in the target by calling malloc(), e.g. ‘int x’ is the same as "x:= *(int *) malloc(sizeof(int))". This is oblivious to the user. The allocated memory is not claimed when a variable is redeclared. Declared variables addresses can be passed to functions and used in other data structures. The keyword ‘T’ must precede user defined types (typedef), e.g. if val is a user defined type, The C code "val *p=(val*) x" becomes "T val *p; p=(T val *) x" in Duel.

Function calls: *func(parm,...)*

Function calls to the debugged program can be intermixed with Duel code. Multi-valued parameters are handled as with binary operators. The call value can have multiple values, e.g. (x,y)() calls x() and y(). Currently, struct/union parameters and return values are not supported.

x,y x..y ..x x..

These operators produce multiple values for single value operands. x,y returns x, then y. x..y returns the integers from x to y. When x>y the sequence is returned in descending order, i.e. 5..3 returns 5, 4, 3. The operator ..x is a shorthand for 0..x-1, e.g. ..3 returns 0, 1, 2. The x.. operator is a shorthand for x..maxint. It returns increasing integer values starting at x indefinitely, and should be bounded by [[n]] or @n operators. ‘,’ retains its precedence level in C. The precedence of ‘..’ is above ‘<’ and below arithmetic operators, so 0..n-1 and x==1..9 work as expected.

x<?y x>?y x>=?y x<=?y x!=?y x==?y

These operators work like their C counterparts but return x if the comparison is true. If the comparison is false, the next (x,y) value is tried, forming the basis of an implicit search.

(x) {x} x;y x=>y

Both () and {} act as C parenthesis. The curly braces set the returned symbolic value as the actual value, e.g. if i=5 and x[5]=3, then x[i] produces the output "x[i] = 3", x[{i}] produces "x[5] = 3" and {x[i]} produces just "3". The semicolon is an operator. x;y evaluates x, ignoring the results, then evaluate and return y, e.g. (i:=1..3 ; i+5) sets i to 3 and return 8. The x=>y operator evaluate and return y for each value of x, e.g. (i:=1..3 => i+5) returns 6, 7 and 8. The value returned by x is also stored implicitly in ‘_’ which can be used in y, e.g. 1..5 => z[_][_] will output z[1][1], z[2][2] etc. The symbolic value for _ is that of the left side value, hence {_} is not needed.
Semicolon has the lowest precedence, so it must be used inside () or {} for compound expressions. The precedence of ‘=>’ is just below ‘,’. Beware that "if(a) x; else {y;} z" is illegal; a semicolon is not allowed before ’}’ or ’else’ and must be inserted before z.

x->y x.y

These expression work as in C for a symbol y. If y is an expression, it is evaluated under the scope of x. e.g. x.(a+b) is the same as x.a+x.b, if a and b are field of x (if they are not, they are looked up as local or global variables). x may return multiple values of different types, e.g. (u,v).a returns u.a and v.a, even if u and v are different structures. Also, the value of x is available as ‘_’ inside y, e.g. x[..100].(if(a) _) produces x[i] for each x[i].a!=0. Nested x.y are allowed, e.g. u.(v.(a+b)) would lookup a and b first under v, then under u.

Aliases: *x:=y*

Aliases store a reference to y in x. Any reference to x is then replaced by y. If y is a constant or an rvalue, its value is replaced for x. If y is an lvalue (e.g. a variable), a reference to same lvalue is returned. for example, x:=emp[5] ; x=9 assigns 9 to emp[5]. Aliases retain their values across invocation of the "dl" command. An alias to a local variable will reference a stray address when the variable goes out of scope. The special command "dl clear" delete all the aliases, and "dl alias" show all current aliases. Symbols are looked up as aliases first, so an alias x will hide a local x.

x-->y

The expansion operator x-->y expands a data structure x following the y links. It returns x, x->y, x->y->y, until a null is found. If x is null, no values are produced. If y returns multiple values, they are stacked and each is further expanded in a depth-first notion. For example, if r is the root of a tree with children u->childs[..u->nchilds], then u-->(childs[..nchilds]) expands the whole tree. y is an arbitrary expression, evaluated exactly like x->y (this includes '_'.)

x@y

The expression x@y produces the values of x until x.y is non-zero, e.g. for(i=0 ; x[i].code!= -1 && i<100 ; i++) x[i] can be written as x[..100]@(code==-1). The evaluation of x is stopped as soon as y evaluates to true. x->y or x=>y are used to evaluate y when x is not a struct or a union. If y is a constant,(_==y) is used. e.g. s[0..]@0 produces the characters in string s up to but not including the terminating null.

#/x &&/x ||/x

These operator return a single "summary" value for all the values returned by x. #/x returns the number of values returned by x, e.g. #/(x[..100]>?0) counts the number of positive x[i]. &&/x returns 1 if all the values produced by x are non-zero, and ||/x returns 1 if any of x's values are non-zero. Like in C, the evaluation stops as soon as possible. For example, ||/(x[..100]==0) and &&/(x[..100]==0) check if one or all of x[i] are zero, respectively.

x#y x[[y]]

The operator x#y produces the values of x and arranges for y to be an alias for the index of each value in x. It is commonly used with x-->y to produce the element's index, e.g. head-->next->val#i=i assigns each val field its element number in the list.
The selection operator x[[y]] produces the yth result of x. If y returns multiple value, each select a value of x, e.g. (5,7,11,13)[3,0,2] returns 13, 5 and 11 (13 is the 3rd element, 5 is the 0th element). Don't use side effects in x, since its evaluation can be restarted depending on y, e.g. after (x[0..i++])[[3,5]] the value of i is unpredictable.

frame(n) frames_no func.x

frame(n) for an integer n returns a reference to the nth frame on the stack (0 is the inner most function and frame(frames_no-1) is main()). Frame values can be compared to function pointers, e.g. frame(3)==myfunc is true if the 4th frame is a call to myfunc, and in scope resolution, e.g. frame(3).x return the local variable x of the 4th frame. frames_no is the number of active frames on the stack, e.g. (frames(..frames_no) ==? myfunc).x displays x for all active invocations of myfunc. As a special case, (frames(..frames_no)==?f)[[0]].x can be written as f.x (x can be an expression).

BUGS

Both '{}' and ';' are operators, not statements or expression separators; "if(x) y; else {z;} u" is illegal; use "if(x) y else {z} ; u". Ambiguities require preceding user-defined types (typedef) with the keyword T, e.g., if value is a user type, C's "sizeof(value*)" is written "sizeof(T value*)", except for the casts "(t)x" and "(t*)x"; sizeof(x) requires parenthesis for variable x.

Unimplemented C idiom include: modified-assignment (x+=y), switch, break, continue, do, goto, scopes, function declarations, initializing declared variables, assignment to bit-fields and register variables, and calling functions with a struct/union parameter or return value. gdb does not store function prototypes, so parameters are not checked.

Gdb itself is buggy, which shows up, especially in symbol tables and calling target functions. Before you report bug, try to do the closest thing under gdb's "print". Send bug to: mg@cs.princeton.edu.

FILES

duel.out tracks duel commands usage. Help analyze duel's use by mailing a copy to mg@cs.princeton.edu.
Duel is available by anonymous ftp at ftp.cs.princeton.edu:/duel.

AUTHOR

Duel is public domain code -- no copy left or right. See the internals documentation for details on porting Duel and using its code. Duel was designed and written by Michael Golan as part of a PhD thesis in the Computer Science Department of Princeton University. I would like to thank my advisor, Dave Hanson, who helped in all phases of this project and to Matt Blaze for his support and useful insight.

Duel stands for Debugging U (might) Even Like, or Don't Use this Exotic Language. Judge for yourself!