A VERY HIGH LEVEL DEBUGGING LANGUAGE

Michael Golan

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

January 1994

To Ela, my love, whom I never had to duel with.

Acknowledgments

I am told that my advisor, Dave Hanson, once said that everything written in this section is "just lies, anyway." I could never tell a lie, so I won't say that Dave showed an infinite amount of patience while working with me. But if it was not infinite, Dave certainly showed a lot of patience, probably more than I deserved.

Dave's comments never cease to amaze me. "There will be no bullets ... bullets are a replacement for proper English" he once told me. At the time, I felt this was extreme. But ever since that day, whenever I take a careful look at a bullet list, I see that Dave was right! Since then, I have listened to Dave's advice very carefully (he also said never to use "very" but I think he will allow it here.) Even when Dave's advice made little or no sense, it has almost always paid off. Thank you, Dave, for sharing with me your knowledge and expertise, and advising me throughout the process of acquiring my degree.

I would like to thank both of my readers, Andrew Appel and Richard Lipton. Dave took most of my poor English out of the thesis first, leaving the readers' red pen almost unused. Nevertheless, both Appel and Lipton have made several insightful comments that helped me to improve and strengthen the overall presentation.

"Granted, Life is too short." said Matthew Blaze while we discussed immortality. Indeed, five years of my life seems to be concentrated into the following few pages of text. I kept thinking about this quote as I woke up every morning (or evening, or afternoon), to put the final touches on this thesis and move on with my life. I would like to thank Matt and many other friends I have acquired in Princeton for their encouragements, friendship, and help. I would also like to thank the department's staff, both technical and secretarial, for making my stay in this department more pleasant than my best expectations.

Princeton University charges \$60 for processing the thesis paperwork, but this amount is less than 0.05% of the overall cost of getting a Ph.D. My work was funded in part by Princeton University and by NSF Grant CCR-9200790, for which I'm grateful.

Last but not least, I would like to thank Ela, my dear wife, for her love and support throughout our Princeton experience. I would also like to thank Yuval, my business partner and genius financial advisor, for his patience and hard work while I was 7,000 miles away. And finally, it goes without saying that I owe a big, special "thanks" to both my parents, for rearing me well enough to pass Princeton's acceptance tests!

Abstract

Programmers spend a significant part of their time debugging, yet most of them simply add print statements and avoid debugging tools. Although the user interface of debuggers has improved significantly in the past decade, their basic debugging methodologies have changed little.

This thesis examines single-process sequential debuggers and attempts to elucidate their weaknesses. Target-state exploration and conditional program execution are essential idioms that cannot be practiced effectively with existing debuggers. State exploration seeks answers to queries such as detecting a linked list containing a negative element and finding where that element was added.

Previous work has focussed on the use of source-language interpreters and graphical interactive exploration as solutions to the state-exploration problem, but these cannot effectively search for a negative element in a linked list, specify a search for it, etc. This thesis describes *Duel*, a very high-level language designed specifically for state exploration in a debugger. *Duel* allows complex state queries to be expressed succinctly and it produces meaningful output. The traditional C operators are extended to process sequences of values, and special operators manipulate these sequences in novel ways. Over a hundred programmers have used *Duel* and their experience substantiates the importance of state exploration and of the inadequacies in previous approaches.

Conditional execution that detects state changes in complex data structures is at the crux of advanced debugging, and it is useful to specify such conditions in *Duel*. Such use of *Duel* raises important semantic and implementation problems. These problems, possible solutions, and *Duel* extensions are discussed.

Contents

Chapter 1

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 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 (Johnson 1978). The basic debugging methodology is still to set breakpoints, run the program being debugged, called the *target*, until a breakpoint is reached, and explore the target state by displaying values of variables. GUIs make this exploration less tedious and more productive, but most are just a veneer over commands that print the values of expressions.

Debuggers are now ubiquitous, but often unused (Gramlich 1983). Many programmers prefer the "add print statements" technique. That simple technique, which is discussed in length in Chapter 2, is often as efficient in locating errors as existing debuggers. Hardware improvements, which have made the compile-run cycle time almost negligible, make this technique even more common.

Some bugs are easy to locate without debugging tools. Other bugs are exposed by the postmortem analyzer available in most debuggers. Many difficult bugs, however, cannot be located efficiently with existing tools, for example, a corrupt pointer might be identified, but how and why it was corrupted cannot. Locating such bugs calls for conditional forward or reverse execution, which most debuggers provide only in a limited way or not at all.

Some of the most difficult bugs present themselves as an inconsistent or unexpected complex data structure or a collection of values, for example, an improperly sorted linked list. Programmers need to compose state and data-probing queries to first find out what is wrong, then compose a condition to be used in conditional execution to find where the error was made. Unfortunately, composing queries about anything but simple variables is either impossible or excruciatingly tedious. Trying to execute the target backward or forward to locate a change in a complex structure, for example, finding when the linked list last became unsorted, is all but impossible.

Programmers interact with most source-level debuggers in the language of the target, for example, C debuggers accept and evaluate C expressions (Linton 1990). 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 (Beander 1983, Johnson 1978).

But source-level expressions are inadequate for exploring erroneous program behavior in all but the simplest cases, for example, displaying only the positive elements of an array. Displaying the whole array, or a single element, is inappropriate. Using source-language statements interactively under the debugger to achieve such tasks is clumsy and tedious — users prefer direct source-code modifications, where the semantics are identical and a standard editor is used, i.e., the "print statement addition" technique.

Using only source-language constructs in debuggers is a major reason for their lack of use. Debuggers must provide better state-exploration mechanisms than possible by adding print statements to the source code; the efforts of this thesis were concentrated towards this goal. The realization of these efforts is *Duel*, a very high-level debugging language, specifically designed for state exploration. *Duel* is based on the view that programmers are best served by a specialized debugging language that is more expressive and flexible than the target's source language.

Duel is an expression-based language derived mostly from C, Icon, a very high-level string-processing language, and, to a lesser extent, from APL and LISP. To display the positive elements of `x[100]` in *Duel*, the expression `x[..100] >=? 0` produces output like `x[43]=92`. A public-domain implementation of *Duel* on top of `gdb` (Stallman and Pesch 1991), a widely-available, traditional source-level debugger for C, is in active use. User experience with *Duel* is discussed in Section 8.1.

Using *Duel* expressions to specify conditional execution would improve the usefulness of debuggers. Such an extension, detailed in Chapter 7, has not been implemented. Many other changes to the debugging environment, as discussed in Section 2.6 and Chapter 7, require further work beyond the scope of this thesis.

Given the dismal state of debugger use, I wanted to not merely plot solutions and implement prototypes, but rather to provide a practical, ready-to-use debugger, usable in existing development environments. Such use also provides immediate feedback on the ideas underlying *Duel*. *Duel* attempts to solve the state exploration issue. Alone, *Duel* cannot make significant change in the use of debuggers. It does, however, provide a basis upon which to build new debugging facilities. Once mastered, *Duel* often renders debugging using print statements tedious and inefficient, enough to increase the regular use of debuggers.

This thesis is organized as follows: bugs, debugging, debuggers, and the importance of state-exploration facilities are discussed in Chapter 2. The design considerations leading to *Duel* are discussed in Chapter 3. The semantics, syntax, usage examples, and implementation are discussed in Chapters 4, 5, and 6. Extensions to *Duel* are discussed in Chapter 7. Finally, user experience and the conclusions are covered in Chapter 8.

1.1 An Overview of Duel

The overall "goal" of debugging is to search the target 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 explorations. Debugging can be viewed as performing queries on a database that contains the target's state (Powell and Linton 1983), so the use of a specialized query-language for debugging seems intuitive.

State explorations occur at various levels, and the above queries are example of the highest. At the lowest level, the values of scalar variables are displayed, and the actual exploration is accomplished by visual inspection, for example, after printing `x` and `y`, the user may find that `x` is not equal to `2*y`. A higher level of exploration would request the evaluation of `x==2*y`, but this request formulates a specific true/false query. The user may prefer displaying `x` and `y` because checking specific values is easier than formulating all cases, for example, `x` might validly be zero for a small `y`. Using only a true/false query is akin to a program proving technique requiring handling of all inputs, while debugging involves only one input set, making the queries simpler to formulate and to test. The state exploration facility must strike a balance between these approaches and be flexible enough to allow mixing of different exploration levels.

Most debuggers can only print the values of expressions, which is of little help in answering all but the simplest 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, for example, 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. Finally, modifying the above query to answer a slightly different question, for example, to check if the list is in ascending order, requires significant code changes. C was not designed for target exploration and adapting it to the task is clumsy at best.

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.

Even a modest extension of expressions evaluation requires managing multiple values, for example, the syntax `x[0..9]`, which naturally represents the 0th through 9th elements of array `x`. Using C-like semantics and syntax to print these elements is too clumsy for interactive queries. Multi-value expressions form the basis for *Duel*.

Duel is an expression-oriented language in which expressions can return a sequence of values with an underlying generator semantics (Griswold, Hanson, and Korb 1981). Operators permit these sequences to be manipulated in novel ways to achieve the goal of concise state exploration, allowing many state exploration queries to be expressed concisely, often as "one-liners" without additional variables or control constructs. The semantics are modeled after Icon's (Griswold and Griswold 1990), but the syntax is quite different.

As a simple example, `x[0..9] >? 1` yields the elements of `x` that are greater than one. 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. Because the evaluation is expression based, it is possible to track the symbolic expression leading to the final value, and *Duel* does so automatically. In the above example, the output might be `x[5]=7`, indicating that `x[5]` is `7`, which is greater than one.

While *Duel* manages multiple values, C's single-value semantics and operators are preserved, so `x[0..99]=0` assigns zero to the first 100 elements of `x`. *Duel* also extends the "`.`" and `->` operators to accept expressions on their right side, which are evaluated under the scope of the left operand, similar to the Pascal's `with` statement, for example, in `hash[..1024].if(val>5) name`, the `if` expression is evaluated under the scope of `hash[0]`, `hash[1]`, . . . , `hash[1023]` — `val` and `name` are evaluated as fields of `hash[`*i*`]`. The equivalent C code is:

```
int i;
for(i=0 ; i<1024 ; i++)
   if(hash[i].val>5)
      printf("hash[%d].name=%s\n",i,hash[i].name);
```

The operator `-->` generates the graph nodes pointed by the left operand using the right expression for expansion. For a linked list `L` threaded by the `next` field, `L-->next` returns each of the elements of the list. For example, finding if list `L` contains two identical `value` elements can be achieved with the query

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

`L-->next` generates each element, the `value` field of which is compared to the `value` field of each of the succeeding elements generated by `next-->next` by expanding the linked list starting with the next element. Compare this compact expression with the C code given on page 4 for the same query. The longer C code hides a bug: the initialization of the inner `for` loop is incorrect. Details of *Duel* are discussed in Chapter 5; a reference manual is provided in Appendix A.

Only a few general-purpose programming languages use generators, partly due to efficiency, but their use as debugging idioms is independent of other usages. Programming and debugging are different tasks: programming is concerned with solving complex problems using carefully written, efficient algorithms. Debugging is concerned with interactive state queries that are quite simple in comparison; efficiency is a minor consideration. Similarly, regular expressions are used in specialized tools, but are not commonly used for general programming.

Duel is designed specifically for debugging C programs, but most of the ideas, even the syntax, are applicable to other imperative languages. C was chosen because it is widely used yet infamous for debugging difficulties. C is the "generic" language in `gdb` and it is supported by most existing debuggers.

Duel is relatively simple — the reference manual contains nine pages including a primer and examples. Its design and implementation appears straightforward, but proved rather difficult. Several semantic approaches were attempted, and the final choice is a mix of several traditional semantics. The syntax, typically an easy issue, also proved difficult. The attempt to preserve the C operators while adding many new operators quickly resulted in an inconsistent and hard-to-use syntax. Syntax changes led to semantic changes, and vice versa. These issues are discussed in Chapter 3. Syntax design is probably time well spent. While not a principal issue, it is the most visible aspect of language design, and often the cause for success or failure.

Finally, the implementation, too, was difficult. Implementing a *Duel* interpreter as an add-on module to existing debuggers and interfacing it directly to `gdb`, a highly complicated debugger, took much time and effort. User experience seems to indicate the effort was worthwhile.

1.2 Why Debugging?

Debugging of simple, sequential programs is important because bugs are likely to continue to slow down software development for the foreseeable future. Yet writing a thesis about simple, sequential debugging is risky; the research community seems little interested in such work. Debugging imperative languages like C is a practical problem, but many programmers and researchers don't use debuggers, and researchers in theoretical areas ignore the subject altogether, leaving a tiny audience for such work.

Research about compilers and programming languages produces rafts of reports about design, theory, experience, and implementation; research about debugging produces only a few. Some believe no research is needed. Close to a million programmers have written C programs, most of which have been successfully debugged with existing tools. Others believe that debugging research should concentrate on debugging of distributed or parallel programs, where "the hard problems are."

A basic, but substantial, problem with debuggers is their use or lack thereof. Ramsey (1993) suggests that part of the problem is inconsistency: debuggers are machine-dependent, each having its own command language and idiosyncrasies. But inconsistency has not stopped users of other tools, for example, C compilers, the so-called "standard library," and the options set for most UNIX utilities.

The basic problem might lie somewhere else. If programmers are not using debuggers, it must be that debuggers do not improve their productivity. This observation is important. Chapter 2 details the capabilities of existing debuggers and attempts to explain it. Because programmers spend a large part of their time debugging (Myers 1980), substantial improvements in programmer productivity cannot be achieved without improvements in debugging. Many commercial products are late to market because tracking and fixing bugs takes too long. The increase in software complexity in the 90's will exacerbate this problem.

The small number of debugger users pose a chicken-and-egg problem. Few commercial vendors strive to improve debuggers because of lack of demand, and users don't demand better debuggers since they don't use them. Part of the problem seems to be that there isn't anything clearly wrong with debuggers, and at the same time, everything with debuggers is wrong. Debuggers are written for use with existing operating systems, hardware, compilers, linkers, and most importantly, existing languages. A breakthrough in debugging requires modifications to all of these, a seemingly impossible task.

In this thesis, I have concentrated on practical solutions. The main result, *Duel*, runs on a variety of systems. The design and implementation of *Duel* tries to cater to a large audience; perhaps it will spark new interest in debugger use and research.

1.3 Related Work

Traditional symbolic source-level debuggers like VAX DEBUG (Beander 1983) and `dbx` (Linton 1990), are separate tools that debug programs written in multiple source languages. They provide mostly low-level debugging idioms like breakpoints and single-stepping, and evaluate only source-level expressions. Although much easier to use than machine-level debuggers like `adb` (Matanzano and Bourne 1977), they provide no new debugging paradigm but merely automate table lookup. `gdb` 4.0 (Stallman and Pesch 1991), on top of which *Duel* was implemented, supports many target machines and operating systems, but uses the same technology.

In the past decade, integrated environments (Ross 1987), graphical displays, and mouse actions (Adams and Muchnick 1986, Cargill 1983, Cargill 1986, Bovey 1987, Russell 1992) have been used to improve the user interface. While making state queries easier, for example, by clicking on a pointer to display the element it points to, no fundamental change in the programmer's ability to assess the target state has been made. Most of these works recognize the need to, for example, display only certain elements of an array, but provide no solutions.

Most recent research has concentrated on parallel and distributed debugging to which complete workshops have been devoted (Miller and LeBlanc 1989, Miller and McDowell 1991); some ideas are also applicable to traditional debugging. Of special importance is reverse execution, and the realization that traveling along the execution path looking for specific states is important. Feldman and Brown (1989) summarized that realization by suggesting typical programmers want to know "when did this tree first get an illegal cycle?". They provide reverse execution to answer the "when" and "first" parts of the question, but no method to describe the "illegal cycle" condition.

Programmers looking for bugs think backwards in execution time and would prefer to use reverse-execution to trace back errors, as discussed by Agrawal, Demillo, and Spafford (1993) who also suggest the use of dynamic slices. Many approaches to implementing reverse-execution have been documented (Balzer 1969, Appel and Tolmach 1990), although Chapter 2 suggests that efficient conditional forward-execution might be easier to implement and often just as useful. Either approach is limited unless a *Duel*-like condition can specify "where to go."

Generalized path expressions, which validate execution paths by matching them to an enhanced regular-expression, have been suggested by Bruegge and Hibbard (1983), Hseush and Kaiser (1989) extended path expressions to support data paths in parallel programs; *Duel* evolved from an attempt to apply regular-expressions to validate and display target data. Event-based debugging, which associates events with program behavior (Bates 1989, Olsson, Crawford, and Ho 1990, Olsson, Crawford, and Ho 1991), uses special languages like EDL (Bates and Wileden 1983) to specify and detect faulty sequences of events.

The Event-based debugging paradigm stresses the execution path leading to a bug. In contrast, *Duel* stresses the importance of an error in a single state. Others have designed new debugger languages based, in part, on similar premises (Johnson 1977a, Olsson *et al.* 1991, Lopriore 1988), and some recent work has focussed on semantic issues (Crawford *et al.* 1992), but none use multi-value expressions or provide *Duel*-like functionality in succinct expressions.

Interactive, graphical exploration of the target state has been implemented in Incense (Myers 1980). Myers discusses the need to provide programmers with "meta data," for example, which elements of an array have non-initial values, but gives no solutions. Nevertheless, Incense touches on the issue of user-specified "interesting" parts of the target state and how they should be displayed, implementing an elaborate system to do so, but not through a specialized language.

Much research has been devoted to making the use of debuggers more palatable. Dynamic linking and incremental compilation (Fritzson 1983) speed up recompilations and treat debugging code and source code equally. Ramsey (1993) implemented a retargetable debugger that uses the compiler to avoid unneeded reimplementations, and provided an identical debugger across multiple platforms. Optimized code can now be debugged, either by hiding the optimization effects (Coutant, Meloy, and Ruscetta 1988) or fully exposing them (Brooks, Hansen, and Simmons 1992).

High-level debugging mechanisms such as conditional breakpoints and watchpoints, which could be more useful in conjunction with *Duel*, are implemented very inefficiently by most debuggers (Linton 1990). Fast code breakpoints (Kessler 1990) and data breakpoints (Wahbe 1992) address these issues with hardware and software solutions. Orders of magnitude of performance improvements are typical. Other hardware-assisted debugging include hardware counters (Cargill and Locanthi 1987, Mellor-Crummey and LeBlanc 1989), and running the debugger on a separate processor (Aral, Gertner, and Schaffer 1989). Johnson (1982) discusses architectural support for debugging.

Sophisticated use of generators, the semantic basis for *Duel*, involving the notion of backtracking and goal-directed evaluation is most apparent in Icon (Griswold and Griswold 1990), a descendent of SNOBOL4 and SL5 (Hanson and Griswold 1978). Budd (1982) has extended C to use generators. Efficient implementations of generators are possible (Walker and Griswold 1992), even with a conceptually clear and flexible implementation (O'Bagy and Griswold 1987).

Chapter 2

Bugs and Debugging

A bug is a programming error that manifests itself during target execution. The target state (data, stack, registers, etc.) is modified with each executed instruction. Normally, these states are *valid*; they are designed to occur during execution by the programmer. A buggy program enters an *error state*, a state unintended or unanticipated by the designer. It then continues in a sequence of error states until it produces an external event indicating a problem — bad output, an exception (core dump), etc.

An external event with an error indication is the *bug result*. The target source location associated with the bug result is the *bug location*. The sequence of instructions executed by the target is the *bug path*. The point along the bug path where the target switches from a valid into an error state is called the *bug origin*, for example, in

```
1   int i,vals[100],x=0;
2   for(i=0 ; i<=100 ; i++)
3      vals[i]=3*i;
4   printf("%d\n",x);
```

the assignment `vals[100]=300` in line `3` is the bug origin. It can cause `x` to be modified, so the output in line `4`, the bug location, is `300`, not `0` — the bug result.

Debugging is the process of exposing the bug path, the error states along it, and of finding the bug location and bug origin. The name "debugger" is a misnormer; a debugger is merely an execution monitoring tool. It cannot "de-bug" the target nor locate bugs by itself. Only the programmer has sufficient information to decide what the actual problem is, for example, whether the condition `i<=100` or the declaration `int vals[100]` caused the problem in the example above.

The bug location is important because it provides the programmer with a known error state. By examining the target state at this location the programmer detects specific problems that help find previous locations of interest along the bug path. This process is iterated until the bug origin is found. In the example above, noting the incorrect value of `x` at line `4`, the bug location, allows the programmer to search backwards to the location where `x` was last set (line `3`).

Knowing the bug origin is not always sufficient. In the slightly modified example

```
int n=100;
int i,vals[100],x=0;
for(i=0 ; i<=n ; i++)
   vals[i]=3*i;
printf("%d\n",x);
```

`n=100` is the bug origin, because, assuming that other parts of the program are correct, it is the first statements to produce an error state, eventually causing the bug result. If `n==100` were a user input, and the only input causing a problem, it would clearly be the bug origin, and therefore plainly known. Nevertheless, programmers require knowledge about the bug path (specifically, the overwriting of `x` at the iteration for `i==100`) before the problem can be fixed. Therefore, the actual tracing of the bug origin, which exposes the bug path, is an important part of the debugging process. Tracing is particularly important when the bug location cannot be identified from the bug result, because no error state is available to examine. In such cases, tracing execution from the bug result allows the programmer to locate a known error state to be examined further.

2.1 What Makes Debugging Difficult?

Given the target states at each point along the bug path and knowledge about the correct target behavior, debugging is a search for points of interest along the bug path and especially the bug origin and location. Several practical issues make debugging more difficult than this formulation suggests.

First, the programmer may not know the correct execution path and states, which is often the case when the error results from interactions between code written by different programmers. Even when programmers understand how the individual parts work, the operation of the combined result might not be understood fully. Software engineering research attempts to address these issues with only a moderate success so far (Brooks 1987).

Second, the sheer number of program states involved makes debugging difficult. Most computer systems do not provide complete target states along the bug path. Instead, programmers must rely on the deterministic nature of software and the ability to reexecute the target to produce the same bug result, investigating the bug path and states along repeated executions. Unfortunately, reexecution can be tedious, or the bug result might depend on nondeterministic interaction with external events (interrupts, network access, etc.) that are beyond the programmer's control. These issues are especially critical in the debugging of real-time, multi-threaded, parallel, or distributed code (Miller and LeBlanc 1989, Miller and McDowell 1991).

The third issue complicating debugging is the introduction of an often subtle effect on the target's execution by the debugging process and the debugging environment. For example, adding a debugging print statements can cause a stray pointer to point to a valid location, resulting in no bug or a different one. Because optimizations must normally be turned off for debugging, subtle bugs that occur only with compiler optimizations are especially hard to find. This issue has been addressed by allowing optimized code to be debugged (Coutant, Meloy, and Ruscetta 1988).

Finally, debugging is complicated by the sheer amount of information in a single state that must be sifted through to determine its validity. Even having reduced the information to several hundreds items or bytes, determining if a state is valid might be as difficult as reprogramming part of the code. For example, determining if a state in an execution of quicksort is valid might be as difficult as writing quicksort.

Once a specific error is suspected, further investigation of locations and states along the bug path is required. Navigating through the bug path and collecting the required information is complex, especially under the conditions described. A programmer-specified method to determine the validity of a state is needed in order to automate such navigation, for example, if the invalidity of a state is traced to a zero-column matrix, how is a search for the location that produced the matrix is specified? Pinpointing the bug origin could be especially hard because the first transition from a valid to an error state is often subtle.

Luckily, the first three issues are not a serious hurdle for most programming tasks. The increase in computing power has eliminated much of the drudgery of rerunning programs; programmers often design and implement their own programs, which they fully understand; programs execute as a single task and external conditions are fully controlled, and subtle side effects caused by debugging are often prevented by disabling compiler optimizations and using a debugger that runs in a separate address-space. Most UNIX utilities and user applications fall into this category. Furthermore, many bugs are easy to detect and fix. Postmortem analysis or a look at the bug result is often sufficient to detect it. For experienced programmers, chasing bugs is a second nature. They often locate bugs by rereading the source and mentally considering the possible problems.

Still, target state exploration makes debugging a time consuming and tedious process. This thesis is concerned mostly with these issues, which are often considered "simple debugging problems" and for which little research has been done. We need to understand fully those "simple problems" and provide efficient tools to cope with them before we can attack the more complicated issues. The lack of research is, perhaps, partly due to the prototype nature of typical programming done by researchers, in which most bugs can be ignored or fixed without a debugger.

2.2 Solving "Simple Debugging Problems"

Besides mentally locating the bug origin by considering all the possible errors in the code, there are two major approaches to the "simple debugging problem." With print statement addition (PSA), programmers guess various error locations and related data items and insert print statements to display them. After recompiling and running the target, the output provides a partial bug-path trace and partial state information at the traced points. With the new information provided, the programmer iterates the PSA process until the problem is located.

The other approach is to use a debugging tool. Often, bug locations like an illegal pointer access are identified by the debugger immediately, providing a known error state with which to begin state exploration. Otherwise, the programmer guesses error locations, inserts breakpoints, and reruns the target trying to detect the bug by looking at the target's state when the breakpoints are reached. This process is iterated until the bug is found. Some debuggers also allow a limited state-search using conditional breakpoints and watchpoints, but these are rarely used due to problems discussed below.

Many researchers seem to assume that the basic low-level facilities as provided by modern debuggers are sufficient (Adams and Muchnick 1986). This assumption has fueled many improvements to the user interface, but little change to basic debugging methods. There are exceptions; Feldman and Brown (1989) discuss the inadequacy of existing debuggers. Three basic mechanisms are required for efficient debugging: bug-path revelation, state exploration, and bug-path travel based on state information.

Bug-path revelation helps the programmer identify critical points along the bug path. The bug location, one point along the bug path, is revealed by postmortem analysis for aborted programs; diagnostic output like that produced by C's assertion mechanism, `assert`, also identifies that point. Execution tracing is used to identify other points by modifying target execution to produce artificial side-effects that clearly identify source locations. The bug-location position along the bug path can be narrowed down as much as needed with this technique.

Given a known error state such as the bug location, a state exploration facility is used to learn why the state is erroneous. Because the amount of information is often staggering, the ability to search and summarize this information is needed. For example, it makes little sense to display all the entries of a large hash table when 90% of the entries are empty.

After understanding the cause of an error at a certain location, the programmer usually wishes to travel back and forth along the bug path to the first or last similar or dissimilar state such as the first state in which the hash table was more than 10% full, or the last location where the hash table did not contain the element `423`. Most likely, the programmer will then explore the new state and reiterate the process.

2.3 Debuggers vs. Print Statements

Nearly all programmers have used debuggers to solve an otherwise intractable problem, yet many regard the debugger as a "last resort" and rely on PSA for most problems. That programmers prefer this low-level facility to a full-featured debuggers is a phenomenon comparable to choosing a line editor over a screen editor. It can be explained by considering how well debuggers and PSA provide for bug-path revelation, state exploration, and bug-path travel.

For bug-path revelation, debuggers excel when the bug result is a core dump, because they contain a postmortem analyzer that displays the bug location. Adding side effects for tracing is accomplished by adding breakpoints when using debuggers; with PSA, the output provides the side effects. Using a debugger is generally considered better because the programmer can interactively decide to explore states and modify the tracing, for example, by single-stepping suspect code.

In many cases, however, hundreds or thousands of points along the bug path are of possible interest, making the interactive nature of debuggers a handicap. For example, a function that outputs text and is called many times is suspected of overwriting the bottom line on the screen in one of the calls. Setting a breakpoint at the entry to this function in order to detect the call that overwrites the last line will result in many futile breakpoints. The PSA "batch" approach is better because it provides complete information about all of the calls in a single output and avoids the hundreds of manual commands needed to continue target execution. It is also common for programmers to turn on and off hundreds of added print statements using macros and to rerun the target in order to gain more information. No comparable feature is available in debuggers.

The state exploration facilities of most debuggers, as discussed in Chapter 1, are inferior to those of PSA. Most debuggers can evaluate only source-language expressions. Complete arrays and structures can also be displayed and interactively explored (e.g., with mouse clicks), but these features are offset by the inability to format, select, and display values as well as possible with PSA. For example, the code to display non-zero elements of `x[100]`

```
int i;
for(i=0 ; i<100 ; i++)
   if(x[i]!=0.0) printf("x[%d]=%10.4lf\n",i,x[i]);
```

cannot be duplicated in most debuggers. Debuggers that provide a full language interpreter offer capabilities similar to PSA, but the full language is not well-suited to state exploration, much less for doing so interactively. In *Duel*, `x[..100]!=?0` displays the same values as the fragment above.

The facility to move back and forth along the bug path to locate specified states is not as often used, but it is invaluable for complex or subtle errors. Consider again the matrix example: the programmer finds a state, s_2, in which a matrix incorrectly contains a zero column. The next natural action is to travel back along the bug path to s_1, the first state in which the matrix no longer contains that zero column. A previous state s_0 without the zero column is not of immediate interest. Alternatively, the desired action is to travel forward from the beginning of the bug path past s_0 to the last state s_1 with a non-zero column before s_2.

Advanced debuggers provide conditional breakpoints and watchpoints, but these are inadequate. Performance is orders of magnitude slower (Linton 1990), conditions like "the matrix has no zero column" cannot be expressed with a simple expression, and forward execution is allowed only to the first place where the condition is true. Specifying travel to s_1 is impossible.

PSA often provides a somewhat better alternative. By displaying the state of the matrix column at various program locations, all of s_0, s_1, and s_2 can be exposed or approximated. The output will contain a great deal of useless information, but s_1 is usually easy to locate. PSA, however, can check and report conditions only at the locations where print statements where added. A global check comparable to a watchpoint is impossible.

2.4 Debuggers and Print Statements, Revisited

Using a debugger is rather similar to using PSA. Debuggers offer a few advantages and have some clear disadvantages and can mostly be considered as an interactive version of PSA. Increases in computing power continue to reduce the edit-compile-run cycle overhead and with it the differences between PSA and debuggers. Furthermore, the inability to step or run the program backwards makes the interactive nature of the debugger limited, somewhat comparable to an interactive editor without a "previous line" command. Also, PSA are available from within the programming language without new learning. Debuggers have a considerable learning curve, and their varying interfaces make changes in the debugging environment excruciating.

Debuggers excel when a limited piece of code contains a subtle bug that only microscopic diagnosis can detect. Such cases make good use of single-stepping and low-level commands that display machine code and registers. These capabilities, however, are not often useful in debugging user applications written in a high-level language. In contrast, PSA excels when a large or repeatedly executed piece of code contains an error related to the values of few variables. PSA can provide a self-contained, if large, execution trace with the variables values at each trace point; the output can then be manipulated with standard tools.

With debuggers, it is typical for a programmer to step through a program, looking for an "interesting state" to investigate further, for example, a certain function called in a certain way. The programmer often does not know the exact number of times the code will be executed before the interesting state appears, nor a simple condition that can be specified with a conditional breakpoint. This methodology often results in repeatedly stepping through code by hand, waiting for the "important event." After several minutes of this kind of tracing, the programmer can easily lose sight of the original target, forget the execution history, or become worried about mistakenly passing that important state. The result is an even slower single stepping, which nevertheless is often futile. Programmers usually prefer to tweak print statements for an hour rather than to repeat the same manual stepping through code for thirty minutes, making PSA appear more productive even when it is not.

The above scenario occurs only because debuggers reverse the way programmers usually think of errors. The natural way to consider errors is to go backwards from the bug result to the bug origin, but debuggers force programmers to trace forward and wait for the error to occur. Bug-path travel and a good state-exploration facility could reverse this situation.

Most debuggers' environments are less friendly than PSA with a modern editor. Print statements can be modified, copied, moved, and hidden using the standard editor commands, and they remain at the same logical position even when the code surrounding them is modified or moved. Only a few debuggers save breakpoints across debugging sessions and none match the versatility of PSA. Because print statements are part of the code, their semantics are unambiguous. In contrast, debuggers often disallow breakpoints in certain locations, or unexpectedly ignore them, for example, in

```
1  for(i=0 ; i<1000 ; i++) {
2     if(x[i]>0) y[i]++;
3  }
```

some debuggers cannot set a breakpoint at `y[i]++` and misinterpret breakpoints on line `3` due to poorly-defined compiler-debugger interactions.

2.5 Shortcomings of Existing Debuggers

There are several other issues regarding the design and implementations of debuggers that directly effect their use, including the user interface, the command language, performance, and bugs in debuggers themselves.

The adoption of full-screen debuggers, despite clear benefits, has been slow. The Blit debugger (Cargill 1983) was developed a decade ago, but some debuggers still have only a command-line interface, for example, Ultrix's `dbx`. Also prevalent are full-screen front ends to command-line debuggers like `dbxtool` (Adams and Muchnick 1986). A full-screen debugger built from the ground up would be better integrated.

Even full-screen and GUI debuggers are still a burden to learn and use. Debugger commands to display and search source code often differ from those of the programmer's editor. The common solution is an integrated development environment (IDE), but IDEs restrict programmers to specific tools and have been largely rejected by the programming community. Even Borland, which sells the most popular IDE for C, provides the best debugging facilities only in its stand-alone debugger. Alternatively, the debugger is run as a subprocess of the editor, like with `emacs` and `gdb`, using the editor as a front end to the debugger. Though potentially far-reaching, most implementations are quite limited, for example, breakpoints are not highlighted in the source.

Debuggers often present a large, but inconsistent and non-portable command set. `gdb` is typical, with over 40 different commands and a thick manual. And yet, many debuggers lack basic features, including single-stepping statements instead of lines, breaking at the last statement of a loop (catching `continue`), undo capabilities, etc. A more coherent approach, using basic idioms as building blocks for higher-level commands has been explored by Crawford *et al.* (1992).

Execution breakpoints, which execute a debugger command rather than stop, can provide results similar to print statements, but not all debuggers provide them and those that do take orders of magnitude longer to execute compared to print statements. Conditional PSA, made possible with standard language facilities, has conditional execution breakpoints as a counter part — also rarely provided by debuggers and often many orders of magnitude slower. The dismal performance of watchpoints and conditional breakpoints makes them nearly useless (Linton 1990).

Debuggers should be carefully designed to produce accurate results. It is hard enough for programmers to deal with their own bugs, let alone a debugger problem. Unfortunately, buggy debuggers are ubiquitous. My limited testing discovered many bugs including treating `-0.0` as non-zero, insisting that a defined symbol is undefined, displaying incorrect variable values, and missing breakpoints. Not all problems are the fault of the debugger but rather a result of language definition and compiler-debugger interaction. Nevertheless, most programmers quickly learn that debuggers' output cannot be entirely trusted, which does not encourage their use.

Finally, debuggers introduce an extra step in the edit-compile-execute cycle. They are often slow to start, and require extra commands in order to provide the target with parameters. Eliminating this overhead is likely to make permanent program testing under the debugger more common.

2.6 Suggested Solutions

The analysis above makes it clear that PSA-like capabilities should be added to debuggers. This addition requires that both the editor and the compiler be modified to allow debugging statements to be added into the source but without requiring recompilation. The tools will cooperate to pass these along to the debugger. A similar approach was suggested by Linton (1990). Tolmach (1992) also notes the usefulness of PSA-capabilities and suggests automating them instead of using a machine-level debugger.

Bug-path travel can be achieved by forward reexecution (FR). A counter of the number of program instructions executed so far is required to achieve this goal, implemented either in hardware (Cargill and Locanthi 1987) or by the compiler (Fraser and Hanson 1991). The debugger would restart the program as needed and track the instruction count to implement automatic search for the *last* state satisfying a certain condition before the current state, providing the same facilities as traditional reverse-execution (RE). Implementing FR is much simpler than RE because complete machine-states need not be saved. RE, often implemented as reexecution from a checkpoint (Feldman and Brown 1989), is really an enhancement of FR. FR is expensive and cannot eliminate the need for RE (e.g., for realtime programming), but for "simple debugging problems" FR provides an excellent, easy to implement solution. FR is also the only likely bug-path travel that can be implemented in complex environments like X-windows, because checkpointing for RE will have to include vast state information that is not readily accessible.

To make the performance of debugging statements comparable to that of PSA, to improve the performance of watchpoints, and especially to facilitate bug-path travel with reasonable performance, new debugging technology must be developed to reduce the overhead of a target-debugger context switches. The current architecture, in which the debugger is a separate process, might be improved enough to provide adequate performance, but I suggest that hardware and operating systems be modified to provide user processes with a debug/run modes, similar to existing kernel/user mode. The debugger would be executed in debug mode so that user code cannot corrupt its state. Because all executed instructions are produced by the compiler, it is sufficient to provide a debug/run switch instruction and to protect part of the address space in run mode. The Intel 386 (Intel 1986) already provides a way to implement such debuggers by using segment protection. Compilers could be modified to call the debugging code at every statement with only a moderate performance penalty for the target-debugger context-switch (Ramsey 1993). Alternatively, the debugger could run on a separate processor and share memory with the target (Aral, Gertner, and Schaffer 1989).

Tolmach (1992) has shown that instrumenting the source code with calls to the debugger, which is executed in the same address space, is possible and quite efficient. His debugger takes advantage of ML's type-safety to protect the debugging code without special hardware. The instrumentation is also used to provide bug-path travel with limited checkpointing, and generalized binary-search is used to provide efficient bug-path travel with monotonic conditions. Tolmach's approach is fundamentally similar to Balzer's (1969), who instrumented the target to produce a complete "history tape" that could then be used to provide debugging information and bug-path travel.

Without improving state-exploration, however, the above improvements, which require significant work and cooperation of many tools, will likely have only a minor effect. Merely providing the same PSA capabilities is unlikely to increase the use of debuggers — people will continue to use the old, familiar PSA techniques. *Duel* or a comparable facility, used for state exploration, bug-path travel conditions, and easy instrumentation of code for tracing (due in part to automated symbolic output), is a necessary first step before other improvements are made.

2.7 Other Debugging Tools

Interpreters provide an alternative debugging environment. Unlike most debuggers, the full language facilities are provided for state exploration. Unlike usual PSA, there is no need to recompile the program when print statements are added. These advantages, however, do not change the overall picture. The same debugging facilities are still needed, and interpreters fare about the same as modern debuggers in this regard. Specifically, better state exploration tools and bug-path travel are as important as for typical debuggers. The term "interpreter" also implies extensive run-time checks, such as the detection of illegal array indexes. Such facilities, though useful, are not tied to interpreters; compiler vendors should provide them, even for C.

Target execution is significantly slower under interpreters. Further, even when execution time is not critical, the "final release" of the code is expected to be compiled, due to the lack of readily-available, redistributable interpreters for C and similar languages. Because interpreters and compilers often use different technology, subtle changes in target execution between these technologies occur. Therefore, use of interpreters as debuggers suffers from the problem of subtle changes to target execution caused by debugging it, an issue discussed in a previous section.

`ctrace` (Steffen 1984), a portable debugging tool, automates PSA. Given C source code, it inserts print statements after every source line, displaying the actual line and related variables. Although it provides a reasonable facility for execution tracing, it is quite limited — the resulting code is not generally useful under a debugger, output is limited to scalar variables, interactions with the C preprocessor can confuse `ctrace`, and various limitations on the use of C idioms like `EOF` are imposed. `ctrace` should be considered an interesting experiment that indicate that similar facilities should be supported directly by compilers and debuggers. Some do; for example, the `lcc` compiler (Fraser and Hanson 1991) provides tracing of function calls.

Dynamic slices (Agrawal, Demillo, and Spafford 1993) use data-flow analysis to locate the possible program statements that have modified a given variable, helping programmers to determine where to set breakpoints. Dynamic slices, however, are currently limited to scalar variables, and do not take into account stray pointers, a common bug. Efficient bug-path travel eliminates most of the benefits of dynamic slices. Furthermore, with efficient watchpoints, an exact slice could be calculated.

Language design influences debugging. Strong type-checking, run-time checks, etc., can automatically detect and report bugs and bug locations. C's design mostly ignores these and other issues related to good software engineering techniques such as modularity.

Many of the common coding errors can be detected by special tools that enforce a more restricted use of the language. The best known tool of this nature is `lint` (Johnson 1977b). Modern lint programs such as `pclint` (Gimpel Software 1986) detect over 300 errors ranging from bad indentation to mixed typing and bad control-flow.

Chapter 3

Design

This thesis takes the view that target-state exploration is central to debugging and that current approaches, like expression evaluations in debuggers like `gdb`, language interpretation in common interpreters, or interactive click-to-expand exploration as in the Blit debugger (Cargill 1983), are inadequate. None of these methods provide a reasonable way to display only the positive elements of a linked-list or a binary-tree, for example.

There are many possible solutions to the target-exploration problem. A language-based tool is only one such solution, and *Duel* is just one such language. A language approach seems justified due to the success of small, special-purpose, languages such as AWK (Aho, Kernighan, and Weinberger 1988), and the inherently more powerful capabilities of a Turing-equivalent language compared with menu-based interfaces. A language also extends the debugger's evaluation of source expressions in an intuitive way.

The design of a new language, even a special-purpose language like *Duel*, requires justification. Some researchers confronted with the idea of *Duel* had strong reservations about the usefulness and need for such a tool, especially compared to a full source-language interpreter. To show that *Duel* is an improvement over an interpreter, special attention was given to operators that replace clumsy C constructs, and many C constructs were added not because they are useful but to allow users to use an interpreted-C style as needed. User choices of operators and constructs can be analyzed to determine user preferences.

While the choice of a language-based approach is easily justified, the choice of specific semantics and syntax is not. Much of the effort of this thesis was concentrated on *Duel*'s semantics and syntax design. What seem like simple and intuitive operators are the result of many design iterations and refinements.

Early design evolved around several simple queries that must be easy to accomplish. Many syntax and semantics decisions were motivated by queries like: "show the positive `val` field in an array of structures `x[100]`, or in a linked listed headed by `y`." The final design uses the syntax `x[..100].val>?0` and `y-->next->val>?0`; the APL and Perl syntax for such queries provide alternative examples.

Even a simple syntactic extension like `x[0..9]` or `x.(a,b)` requires the management of multi-value expressions, for which several semantic models exist, for example, lists and arrays. A related semantic issue is the production of output useful for debugging, similar to that which users manually produce with print statements addition.

3.1 Background

The inspiration for *Duel* was the demonstration of the illustrating compiler (Henry, Whaley, and Forstall 1990). After seeing complex data structures presented graphically for correct and buggy programs, I wanted to implement an illustrating debugger.

The illustrating compiler interpreted data structures and source code to automatically decide on a graphical representation and displayed all of the program's data. This scheme is quite limited and works only on simple programs. C code using, for example, unions to store pointers depending on context, cannot be automatically interpreted. Also, with large amount of data, users must choose a subset to display.

With debuggers, programmers must describe data structures to be displayed and the interactions between them. Myers (1980) displayed data structures graphically under the debugger in Mesa, a language similar to C but with less type-system "holes." Only minimal user assistance is required, but users are unable to display, for example, only the last few elements of a linked-list. His system also seems to produce misleading pictures for large data structures, for example, a pointer to the tail of a long linked-list might appear pointing to another element.

Originally I thought of using regular expressions, similar to those used in the programming language AWK, to describe which data to display, for example, using `x(->y)*` to match `x`, `x->y`, `x->y->y`, etc., and `a[0..99]` to match `a[0]`, `a[1]`, ..., `a[99]`. But after toying with this idea for a short while, it became clear that such a description could be used to query data structures independently of a graphical display. This observation led to the realization that state exploration under existing debuggers is almost impossible and to the development of *Duel*.

3.2 Considerations

Duel is designed to work in existing environments with existing tools. Tools that require changes to the programming environment are unlikely to succeed due to the complexity involved and rejection by users who, while willing to accept a new or improved tool, refuse changes to the environment or the way they work.

Duel must be better than just the use of C code for state explorations. Interpreters (Ross 1987) and some debuggers (Russell 1992) already provide C constructs as means for program state exploration, but C is poorly suited for such a task. C is a general-purpose programming language that stresses efficiency. *Duel* stresses program state exploration; efficiency is secondary.

A prime concern is the ability to pose non-trivial state queries concisely without resolving to the use of explicit control structures (such as `for`) and debugger variables. Imperative control structures provide explicit execution control, but require precise planning and coding and can introduce bugs into the debugging code. Users end up concentrating on writing the query instead of the original purpose.

Because the results of queries are often unexpected (due to bugs), users should be confident that queries work as expected. Long or complex queries cannot provide that confidence and cause users to recheck the queries to verify that the debugging code itself has no bugs. Imperative control structures, however, should not be ignored. Users understand their use and can use them when other solutions are impossible or unnatural.

It is not enough to make the formulation of typical queries easy. It should also be possible to easily adapt one query into a similar one without a complete rewrite, for example, to convert a query that displays certain elements into one that counts their number. Debugging is partially a guessing game. The easier it is to check a guess, the faster users can orient themselves towards the bug origin.

The syntax and semantics of types, values, evaluation order, and operators are fundamental to the source language. The query language must provide for the evaluation of simple source-expressions with the smallest possible deviation from the source. For example, if `x[100]` is an array, C programmers expect `x+1` to be the same as `&x[1]`. If the design uses `x` to represent the list of `x`'s elements and `x+1` the list of each element plus one (`x[..100]+1` in *Duel*), users will be confused. Most debuggers don't follow the precise semantics of C. For example, adding an `unsigned short` to a `long` produces an `unsigned` result in `gdb`, contrary to the ANSI C standard.

Operations whose semantics are undefined, yet possible, should be allowed. The results, whenever possible, should be identical to those in the current implementation, for example, displaying the value of `x[11]` for the array `x[10]`. It should also be possible to manipulate bad, partial, or inconsistent data structures, for example, to display the first three elements of a linked list, when the pointer to the fourth element is invalid. Therefore, it is unreasonable to use semantics that require retrieving all of the list's elements before selecting some of them.

Even though the speed of the current implementation was not a major consideration, the maximum achievable performance for a particular semantics is. *Duel* extensions for watchpoints and conditional breakpoints as described in Chapter 7 require a fast implementation. Semantics that require, for example, complete retrieval of a large array just to select a few elements cannot achieve high performance.

Users often require extensive output from a query, for example, the display of complete or partial data structures. The query language should provide, automatically when possible, output suitable for debugging. For example, a query to display the positive elements of array `x[100]` (`x[..100] >? 0` in *Duel*) should display both the elements and their indexes, for example, output like `x[43] = 21`. Similarly, the output of certain linked-list elements should indicate their location in the list. The output should also be in a form suitable for use in another query, for example, *Duel* displays `head-->next[[5]]->val = 9` to indicate that the 5th `val` element of the linked list `head-->next` is `9`. *Duel* also accepts similar expressions as input.

The basic operations that should be supported by a query language are composition, selection, grouping, and projection. Composition allows the result of a subquery to be used in another query, for example, to use the elements of one array as indexes into another. Selection allows specific elements to be selected, for example, those that are positive. Grouping replaces a set of values with a single value, for example, their count. Projection allows extraction of individual components from a single value, for example, a certain field of a structure. *Duel* attempts to make the integration of these operations seamless and provides only minimalistic operators to achieve each goal, relying on a combination of simple operators to accomplish more difficult tasks.

Finally, a language whose sole purpose is to debug other programs must be easy to learn, use, and relearn. It must strike a balance between usefulness and compactness. Idiomatic use is expected; care must be taken to minimize unexpected results from such use.

3.3 Alternative Semantics

Programmers often wish that the debugger's print command could handle array subranges, for example, syntax like `x[30 to 40]`, `x[30:40]`, or `x[30..40]` to display the elements `x[30]`, `x[31]`, ..., `x[40]` of array `x`. This seemingly simple extension to the print command raises many semantic issues, for example, to what does `1+x[30 to 40]` evaluate?

Such semantics must manage multi-value expressions and operators — expressions that return more than one result and operators that operate on such expressions and return multiple results. The original intention was to treat `x[1..10]` as a shorthand notation and expand it symbolically before evaluation begins, but this approach is too limited. Many semantic models to handle multi-value expressions already exist. The most common approaches are generators, arrays, and lists, possibly combined with lazy evaluation. *Duel* uses generator semantics.

List semantics, such as in LISP, are quite common. An interesting list-semantics language that also allows for side effects and C-style expressions is Perl (Wall and Schwartz 1990). In Perl, `1..5` is the list of integers `1` though `5`. Arrays can be indexed by a list, for example, `x[1..5]` is the list of elements `x[1]` through `x[5]`. The benefits of these semantics are clear: they are easy to define, implement, and understand. The disadvantage is in the way operators are defined.

If `P` and `R` are lists, `k` is a scalar, and `(k)` is a list of a single element `k`, the expressions `P+R`, `P==R`, `P+k`, `P==k`, `P+(k)`, `P==(k)`, and `(P,k)` need semantics assigned to them. With list semantics users expect `P==R` to compare the lists element-wise (returning a scalar), `P+R` to add the lists element-wise (returning a list), `P==k` to compare each element to `k`, and `P+k` to add `k` to each element (returning a list). Therefore, `(k)`, the list of a single element and `k`, a scalar, are different, in turn making operators that return scalars and operators that return lists different. Perl suffers from this confusing issue. The alternative — treating `k` as a one element list (and eliminating scalars) as in APL — is discussed below.

There are other serious problems with list semantics; unless special optimizations are applied, expressions like `1..100000` generate a list of 100,000 elements even if not "used," and expressions like *Duel*'s "`0..`" are impossible. Similarly, if all the elements of a linked-list are fetched into a "list" before further processing, bugs in the linked-list pointers would prevent processing altogether.

Lazy evaluation (Henderson and Morris 1976) has been used in functional languages to allow the incremental generation of infinite sequences, and it could be used in a state-query language. Lazy evaluation combines well with partially-defined or buggy structures and the general list semantics, for example, `p[0..9]->val` would evaluate `p[`i`]->val` only when its value is needed. Lazy evaluation, however, does not fit well with the usual C semantics, because suspended evaluation can produce unexpected results different than those expected by C programmers when side effects are "skipped." Furthermore, C's imperative semantics clash with lazy evaluation, especially when pointers and aliasing are involved. In contrast, generators have a well-defined evaluation order that can co-exist with C idioms.

Array semantics are similar to list semantics, but common operations are accomplished differently, for example, the "`+`" operator is defined only for equal-size arrays. APL (Iverson 1962) uses array semantics. To add the scalar k to each element of array A of size n, a temporary array of n elements, each k, is created and added to A. To get the positive elements in A, it is compared with an array of zeros and the result, an array of boolean values, is used to select the positive elements of A.

APL semantics are succinct and useful, but they require a different mindset. C programmers are used to loops and element-wise comparisons, not to simultaneous manipulations of whole structures. APL is especially strong at mathematical computations and multi-dimensional arrays, but neither is required in the debugging of typical C programs. Finally, array semantics suffers from the same problems as list semantics, for example, `1..100000` generating too many elements. As with lists, lazy evaluation can overcome most of these problems, but it also introduces unexpected results when combined with C.

3.4 Generator Semantics

Duel is based on generator semantics (Griswold, Hanson, and Korb 1981). The problems with other semantic models may make this choice seem clear, but the original design used lists, not generators. Nevertheless, the practical differences between generators and lazily-evaluated lists or arrays is small. Generators provide a more consistent evaluation model with an evaluation order that is easier to understand.

The goal-searching mechanism of generators seemed unnatural to C programmers. Using `x[..100] >? 0` to display the positive elements of `x` is intuitive, but Icon (Griswold and Griswold 1990) semantics for `if` and the overall emphasis on "goals" is not. As the design of *Duel* progressed, however, the semantics gradually incorporated full use of generators. Generators, however, are still a tradeoff: syntax like `x[..100]+y[..100]` for adding two arrays element-wise is impossible.

Generators provide a natural searching mechanism. Single-valued generators reduce to C-like semantics; side effects are easy to combine and apply; lvalue and rvalue C semantics can be preserved, and the "source" of the generated values is easy to track. Generators are not without their shortcomings. They interact poorly with C's `&&` and `||` operators, and sometimes have unintuitive semantics compared with list or array operations. For example, `x[..100]=y[..100]` does not assign `y[i]` to `x[i]` for $i = 0 \ldots 99$. Instead, it assigns each `y[j]` to every `x[i]`, having the same effect as `x[..100]=y[99]`. Such problems are easily explained, and achieving the intended operation is simple, for example, "`i:=..100 => x[i]=y[i]`." The success of Icon indicates these problems are minor.

In Icon, all expressions can produce zero, one, or more values. Its goal-directed evaluation seeks the first "successful" result by trying all possible combinations of values generated by each subexpression. For example, `(1 to 5)` produces the integers 1 through 5, and `(2|4)` produces the integers 2 and 4. Expressions like `x=(2|4)` "succeed" if `x` is 2 or 4, and are often used in `if` statements, for example, `if(x=(2|4))` in contrast to `if(x==2 || x==4)` in C.

While *Duel* shares with Icon the same underlying generator semantics, *Duel* does not use goal-directed evaluation and is therefore quite different; typically, Icon and *Duel* code to achieve similar tasks shows little resemblance in syntax and style. In *Duel*, all of the alternatives for an expression are evaluated, unless special operators ("`@`" and "`x[[y]]`") halt the evaluation. In Icon, only the first alternative is evaluated, unless special operators are used. Because *Duel* supports the full range of C operators, common C idioms such as "`if(x>0 && x<10) x`" are usually used, even though the Icon-style *Duel* code `(x >? 0) <? 10` is semantically identical.

With generators, $e_1 \times e_2$ for most binary operators $\times$ has the usual single value C semantics when both e_1 and e_2 return a single value. When either expressions (but not both) return multiple values, generators apply the operator to each pair of elements, which is what most C programmers expect. When both expressions return multiple values, generator semantics produce a "cross product," which can confuse novices.

Generators and their efficient implementation are well documented (Budd 1982, Griswold 1982, Griswold and Griswold 1986). With generators, only values that are required are fetched, making the creation of infinite sequences like `1..` natural. The processing cost of `0..1000` and `0..10` is the same if only the first 10 values are used.

3.5 Syntax and Semantics

Duel's syntax and semantics are described in detail in the following chapters. Most of the specific choices reflect the discussion in Section 3.2.

Duel uses C types and values, and has no scoping rules of its own except aliases. *Duel* also uses C's syntax and semantics for all the C operators except comma. Programmers know the details of these semantics and syntax, and deviation from them is likely to be confusing. Furthermore, I have found no need to modify the C semantics while recasting them as generators. C's richness in operators does leave only a limited choice for *Duel* operators, but introducing, for example, "`>?`", and keeping C's "`>`" semantics unmodified was preferred to changing the semantics of "`>`".

Most of the new operators in *Duel* can be traced back to either similar operators in Icon or similar programming idioms in C. For example, the C idiom to iterate over the string `str`, "`for(s=str ; *s ; s++)`", inspired the `stopat` operator, and Icon's `every` inspired the `imply` operator.

Duel extends C's "`.`" and "`->`" operators into scope resolution operators. Just as `(x[1],x[2])` can be rewritten as `x[1,2]`, it is possible to rewrite `(y.a,y.b)` as `y.(a,b)`. This seemingly simple extension has far reaching implications, including the need for dynamic scoping. Because *Duel* uses dynamic scoping, variable names are looked up only when they are evaluated; the same name could reference fields in different structures, i.e., in `(x,y).a`, which evaluates to `x.a` and `y.a`, `x` and `y` need not be of the same type. Dynamic scoping allows expressions that "make sense" but that are not allowed under static scoping.

Aliases, another novel feature of *Duel*, are designed to subsume debugger variables. They require no type declaration and can store both lvalues and rvalues. Space for aliases and constant strings is allocated in the target, so if their address is taken or passed to a target function, C semantics are preserved. Russell (1992) tried to avoid this issue by allocating space in the debugger address space, leading to complications. Target-space allocation is done by calling the target's `malloc`, which must behave like the standard-library function. `gdb` uses a similar approach.

Other operators provide succinct ways to manipulate linked-lists, trees, and general graphs. They eliminate the need for explicit recursion for most complex data structures, and provide indexing of elements.

The target name-space is polluted by *Duel* aliases and reserved names like `frame`. The alternative, separate name spaces, for example, by forcing all *Duel* names to begin with "`$`", is easy to implement but seems undesirable, because programmers often forget the "`$`", resulting in syntax errors or a mistaken reference to target names. The pros and cons of such choices are discussed by Crawford *et al.* (1992).

Because *Duel* uses expressions instead of statements and dynamic rather than static scoping, some syntax differences from C, as documented in Section 5.2, are unavoidable. Most noticeably, user-defined types must be preceded by the keyword `T` so that the parser can distinguish between variable and type names. Casts of the form `(t)x` and `(t*)x` for a user-defined type `t`, however, have been added after initial user experience. Although these nuances result in cumbersome syntax rules, users rarely encounter problems.

3.6 Symbolic Output

Most imperative languages provide little, if any, support for output of symbols in conjunction with the values they represent. Users often use special formatting to display the related symbols, for example, `printf("i=%d x[i]=%d\n",i,x[i])` produces output like `i=3 x[i]=5`. One of the goals of *Duel* is to eliminate the use of print statements as a debugging tool, so it is important to provide such output automatically, i.e., if the 3rd element of array `x` is 5, the output of queries like `x[..100] >? 4` should be "`x[3]=5`".

APL provides mechanisms to extract the indexes of the positive elements of `x`, but they require array-style operations; it is also overkill for most queries. Originally, *Duel* expanded expressions symbolically before they were evaluated, so each printed value had an associated symbolic expression. With generators, however, the "correct" symbolic output is unclear.

The solution in *Duel* is to evaluate a symbolic result in addition to the actual result of each operator. This simple idea leads to high-quality output in most cases, but sometimes, for example, with temporary variables, the symbolic result shows a name rather than a value. The "`{x}`" operator solves this problem by setting its symbolic result to the actual result. Also, with some operators the symbolic result is a compromise, for example, "`>?`" produces only the left operand's symbolic value. In some queries, the right operand's symbolic value is important. The alternative, keeping both symbolic values, provides poor output in most cases. Special operators to further manipulate symbolic values have also been considered, but seem rarely useful.

3.7 Other Programming Languages

The ideas of *Duel* should be applicable to most imperative programming languages, for example, Pascal, Fortran, Modula-3, etc., but object-oriented programming languages and languages that support operator overloading provide new challenges. For example, if the indexing operator x`[`i`]` is overloaded, what should the semantics of `x[1..10]` be? Indexing `x` with an iterator class provides further complications.

Languages like LISP and APL seem well fit to debug code written in the same language, but some weaknesses are apparent. There is no access to the active procedure calls and their local variables, so it is impossible, for example, to determine if the recursive procedure `factor(n)` appears on the stack with a negative parameter, a task that can be easily expressed in *Duel*. Displaying output useful for debugging with these languages is also hard, for example, to print the elements of `x[100]` that are positive, together with the indexes, is non-trivial. A *Duel*-like extension to these languages is therefore worth while, although syntax and semantics are likely to be different.

Chapter 4

Semantics

Duel is an expression-oriented language in which operators produce a sequence of values based on the generator semantics of Icon (Griswold and Griswold 1990). For example, `5+(2,3)` produces the values `7` and `8`. Operators like "`,`" produce multi-value results from single-value operands. Others, like "`+`", produce multi-value results only from multi-value operands. These semantics are difficult to describe using traditional descriptive techniques because each operator's description depends on its operands — there are no "atomic" operators.

This issue was evident with Icon and has been addressed by describing Icon's operational semantics with an Icon-like pseudo language (Griswold 1982); a similar approach is used here. *Duel*'s semantics are described operationally using a C-like pseudo-language that mirrors a possible implementation and manipulates an abstract syntax tree (AST). 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 device. C's semantics are relied upon in describing *Duel* operators. Manipulating ASTs instead of a concrete syntax emphasizes the importance of the semantics and its independence from the syntax, which is described in Chapter 5.

4.1 State-driven Description

Duel evaluates an expression by traversing its AST recursively. All AST nodes have an `op` field, which identifies the node's operand, and a `kids` field, which is an array of pointers to the operand nodes. Nodes for specific operators have additional fields, for example, a node for a constant includes a `constant` field that holds the constant itself. ASTs are specified by a simple LISP-like notation, for example, the AST for the expression `a*5 + *b` is

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

If all expressions returned only one value, evaluation would use a standard tree traversal, as in the `eval` function

```
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 possible values and tracks the current type. `apply` uses the C semantics to evaluate the result for each operator; it uses the type information to check, fetch, and convert values.

Duel expressions can produce more than one value, for example, `(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 by reevaluating the right-hand operand `(5,9)` for each value produced by the left-hand operand and producing the sum of the left and right values.

This feature complicates only `eval`, not the code for each 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` reevaluates 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 b1
3  b0: n->state = 0
4      n->value = eval(n->kids[0])
5      if (n->value == NOVALUE)
6        return NOVALUE
7      n->state = 1
```

```
8  b1: u = eval(n->kids[1])
9      if (u == NOVALUE) goto b0
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 reevaluating `(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, in which two procedures run independently, one acting as a consumer processing values generated by the other, the producer. Coroutines are similar to, but pre-date, non-preemptive threads (Knuth 1973).

4.2 Coroutine Description

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 produces e and preserves enough information for the computation to resume after the `yield` statement and to produce the next value. In effect, this state preservation replaces the explicit manipulation of `n->state`. For example, `(alternate` e_0 e_1`)` produces all of the values of e_0 followed by the values of e_1. Its state-driven description is

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

The simplified coroutine description 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, for example,

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

4.3 Specific Operators

Most of the unary operators manipulate and produce a single value at a time. They are defined by

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

The generator `(to` e_0 e_1`)` produces the integers from e_0 to e_1 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, for example,

```
(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, for example, `(ifgt` e_0 e_1`)` produces e_0 only if e_0 is greater than e_1. The semantics are

```
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 `oror` illustrate the problem.

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

e_0 `||` e_1 produces `1` for each non-zero value of e_0; for each zero value of e_0, *all* of the values of e_1 are produced with non-zero values replaced by `1`. When e_0 and e_1 are single-value expressions, these semantics are equivalent to C's. With multi-value operands, confusion can arise, for example, `(7,0,0) || (5,0)` produces `1`,`1`,`0`,`1`,`0`. Semantics to produce only a single `0` or `1` value for `||` are possible, but run counter-intuitive to the semantic of other C operators like "`&`" and "`|`".

Several operators provide execution control flow. The operator `(if` e_0 e_1 e_2`)` evaluates e_0; for each non-zero value of e_0, it produces all of the values of e_1, and for each zero value of e_0, it produces all of the values of e_2

```
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
```

These semantics depart from Icon's "`if` e", in which e is a "bounded expression," evaluated for a single result, producing traditional control-flow, i.e., only one of `then` and `else` is executed, exactly once. The semantics above are more appropriate for `if`-expressions and short state-exploration commands.

A sequence of expressions, `(sequence` e_0 e_1`)` evaluates e_0 but discards its values, and then produces the values of e_1.

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

The related operator `(imply` e_0 e_1`)` is similar, but produces e_1's values for each value of e_0, like Icon's `every`

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

When e_0 is a `to` operator, `imply` is a loop over e_1, for example,

```
(imply
   (to (constant 1) (constant 10))
   (constant 27)
)
```

produces 27 ten times.

Finally, `(while` e_0 e_1`)` produces e_1 only if *all* of the values of e_0 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_0 is a single-value expression. Notice that once e_1 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_0 e_1`)` produces the elements of e_1 given by the integers in e_0. 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_1's `state` is reset so that it starts anew for each value of e_0. The actual implementation of `select` avoids the reevaluation of e_1 when e_0 returns increasing values by continuing the previous evaluation of e_1. The operator, `(stopat` e_0 e_1`)` produces the elements of e_1 but stops as soon as all the results of e_0 are non-zero.

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

The check of e_0's values is similar to that in the `while` operator, but `while` reevaluates its "body" e_1 in every iteration; `stopat` evaluates e_1 only once and produces one value at each iteration. As an example, `stopat` can be used to generate the characters of a string until the null character is found.

Several "reduction" operators reduce a sequence of values to one value, for example, `(count` e`)` returns the number of values produced by e, `(andall` e`)` logically-ands all the values produced by e and returns `1` or `0`, and `(equality` e_0 e_1`)` returns `1` if the values produced by e_0 are equal pairwise to those produced by e_1 and `0` otherwise.

```
case EQUALITY:
   while (u = eval(n->kids[0])) {
      if (v = eval(n->kids[1])) {
         if (u != v)
            return 0
      } else
         return 0
   }
   if (eval(n->kids[1]))
      return 0
   return 1
```

The last two `return 0` statements account for the case when the lists are of unequal lengths.

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. Single-argument functions are treated like C unary operators, double-arguments functions like C binary operators, etc. For example,

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

prints

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

4.4 Scopes and Expansion

As suggested above, `(name "`x`")` fetches the value of the variable x using `fetch`. x can be a variable in the target program or a *Duel 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, for example, 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
   }
```

Defined aliases are added to the aliases table by `alias` and remain there until the *Duel* `clear` command is issued. Aliases mask program variables of the same name because `fetch` looks up names in the aliases table first.

The operator `(with` e_0 e_1`)` evaluates e_1 in the "scope" of e_0. When e_0 is a structure, "opening the scope" of e_0 makes the fields visible as ordinary identifiers. Names in e_1 refer to the appropriate fields in e_0; 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])) {
      pushscope(u)
      while (v = eval(n->kids[1]))
         yield v
      popscope()
   }
```

The `pushscope` and `popscope` functions manipulate a name-resolution stack in which `fetch` looks up names as fields of pushed structures before they are looked up as aliases or target variables. Also, the special name "`_`" in e_1 refers to the top stack-value, i.e., the latest value produced by e_0. The name-resolution stack is also used by other operators, for example, `stopat`, to produce similar "scope opening" effects by pushing the left-hand operand on the name-resolution stack before evaluating the right-hand operand.

To traverse data structures, `(dfs` e_0 e_1`)` "expands" the data structure e_0 using e_1 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_1`)` are stored in list `L`, then pushed in reverse 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, `pushscope` and `popscope` manipulate the name-resolution stack described above, and `listnew`, `listappend` and `listreverse` manipulate lists in a standard way.

```
case DFS:
   while (u = eval(n->kids[0])) {
      stack(n, u)
      while (v = unstack(n)) {
         pushscope(v)
         listnew(L)
         while (w = eval(n->kids[1]))
            listappend(L,w)
         stack(n, listreverse(L))
         popscope()
         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 reversal of the values generated by e_1 before they are stacked makes sure that nodes are visited in the expected order, i.e., `left` is generated first and pushed last so it is also visited first.

4.5 Values, Types, Constants and Names

Value semantics adhere to the ANSI C standard (Ansi 1990), including type promotion, conversion, arithmetic, and the definition of rvalues and lvalues. There is no required compile-time type checking in *Duel*. Compilation is limited to converting the input into an abstract syntax tree and to verifying that type names are valid, for example, in "`(employee*) x`", `employee` is checked to insure it is a valid type name, but no checks of `x` are made, which is stored in a `name` node. Types are checked at run time for each application of an operator. Those checks also obey the ANSI C standard, for example, comparison of pointers of different types is disallowed unless one is a `void` pointer.

Constant strings are allocated in the target's address space. As in ANSI C, their value has type `char*`. *Duel* also allows variables to be defined. They are allocated in the target's address space and initialized to zero.

Duel adds one special value type, *fvalue*, that references an active function call on the stack frame by an integer value indicating the stack depth (zero being the top of the stack). Fvalues are created from integers with the `frame` operator. They can be compared to each other and to function pointers; the comparison result is `1` if the given frame is an activation of the given function and `0` otherwise. Fvalues may also be used with the `with` operator to "expose" the local variables of that frame in the scope of the second operand.

Several *Duel* operators are simple compositions or variations of other operators, for example, the `for` operator is a combination of the `while` and `sequence` operators and `arrow` ("`->`") is a simple combination of the C unary "`*`" operator and the `with` operator. A detailed reference-manual for *Duel* appears in Appendix A.

4.6 Symbolic Evaluation

The `eval` function actually returns a pair: the computed value and a *symbolic* value. The symbolic value is a string that represents the "source" of the computed value. Operators combine the symbolic values of their operands using simple string manipulations to produce their symbolic value. For example, if `x[7]` is `5`, then `2*x[4+(0..6)] >? 0` outputs `"2*x[4+3] = 10"`.

The symbolic value returned for a symbol is the symbol as a string. For a constant, the original user input for that constant is returned. For a binary operator $a \times b$, if `A` and `B` are the symbolic values of the operands, the symbolic result is the string `"A×B"`. Unary operators are similar. The `to` operator returns a symbolic value that represents the generated value, for example, `1..5` returns `"4"` as the symbolic value when 4 is returned, etc.

Most operators that return an unmodified value return its symbolic value also unmodified, for example, `x >? 0` returns the value of `x` and its symbolic value `X` if `x` is positive. Several operators manipulate their operands in a special way. The `dfs` operator `x-->y` evaluates the values of `y` with the symbolic value `"X->Y"`. Multiple occurrences of `Y` in the result are replaced with a shorthand notation, so that instead of `"h->next->next->next"`, the symbolic result is `"h-->next[[2]]"`, etc. The `count` operator, `#/x`, returns the symbolic value `"#/(X...)"` where `X` is the first symbolic value returned by `x`. The "`_`" as a reference to "self" returns the symbolic value it refers to, not `"_"`. The `with` operators are careful not to produce a `"X.X"` result.

Finally, the `(symval` e`)` operator allows users to manipulate symbolic results. It sets the symbolic value to the string that represents the computed result of e, for example, the symbolic result of `(1+2)` is `"(1+2)"`, and the symbolic result of `(symval (1+2))` is `"3"`. *Duel*'s top-level evaluation mechanism prints the resulting pairs of computed and symbolic values unless they are identical in which case only the computed value is printed.

Chapter 5

Syntax

Providing a concrete syntax for the semantics presented in Chapter 4 is a challenging task. Since *Duel* is used to explore the state of C programs, it is useful to provide a superset of C operators and their semantics, but C already provides a rich set of operators, leaving few "free" operators that are unambiguous and easy to type. The syntax presented below is the result of many changes and the consensus that comes only from actual experience. The final result includes all the C operators; casts control structures, like `for`, `if`, etc. as expressions instead of statements; and uses "`#`", "`@`", and "`?`" to specify numerous *Duel*-specific operators. For the C operators, only the semantics of "`,`" have been modified substantially.

Duel is implemented as an extended version of `gdb`. The following sections introduce *Duel*'s syntax with examples that reproduce the use of *Duel* under `gdb` using the `duel` command; a reference manual appears in Appendix A. The `duel` command accepts expressions, compiles them into ASTs, evaluates them, and displays the results in the format `X = x` for symbolic value `X` and value `x`. Displays like `1 = 1`, however, are suppressed. Symbols are looked up at evaluation time using dynamic scoping. The scoping rules for local, static, and global variables depend on the debugger's lookup algorithm.

5.1 Details

In the absence of generators, *Duel* expressions are essentially equivalent to a debugger's "print" command, for example,

```
gdb> print 1 + (double)3/2.0
2.5
gdb> duel 1 + (double)3/2.0
1+3/2.0 = 2.5
```

As this example suggests, the `duel` command provides an extended version of `gdb`'s `print` command. *Duel*, of course, is not limited to single-valued expressions. All the results of the `duel` command's expression argument are produced and displayed, for example,

```
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
```

This example omits the symbolic output, which is described later. The comma operator is the concrete syntax for the `alternate` operator described in the previous chapter; e_1`,`e_2 produces all of e_1's values followed by e_2's values. The operator "e_1`..`e_2" specifies the `to` operator; it produces the integers from e_1 to e_2 inclusive, counting downwards if $e_1 > e_2$. The "`..`e" operator is shorthand for `0..`e`-1`, and "e`..`" is a shorthand for e`..maxint` where `maxint` is the largest integer value. It effectively generates an infinite sequence.

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[..1024]->scope = 0 ;
```

clears the `scope` field of the first `symbol` on each list. `hash[..1024]` produces lvalues; the semantics of C's assignment are unchanged. A terminating semicolon causes the expression to be evaluated for side effects only and to produce no output.

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 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, for example,

```
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`, not just the values that are `7`.

All of the C comparison operators have a *op*`?` counterpart, for example, `!=?`. *Duel*'s constants and type promotions are also identical to C's, so `0` is the null pointer, 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: being a superset of C, 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, for example, in debugging and maintenance. *Duel* expressions, however, are often ephemeral, being written once and read at most once. *Duel*'s syntax is designed to facilitate, on-the-fly, left-to-right composition, for example, `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, for example, the C (and *Duel*) code for the search just described is

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

In `gdb`, commands must be entered on a single line; long lines have been wrapped and indented here for convenience. *Duel* declarations, for example, `int i`, establish aliases to newly allocated target locations, having the same effect as a variable declaration in the target program.

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

gdb> duel i:=..1024 => (hash[i] !=? 0)->scope >? 5
```

As suggested by its semantics, *Duel*'s `if` is an expression, for example,

```
gdb> duel int i; 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 symbol for all symbols, even aliases. Enclosing an expression in braces, which otherwise act like parenthesis, overrides the default display for that expression and causes its value to be displayed instead, for example,

```
gdb> duel int i; 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, for example,

```
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, for example,

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

The value of e_1 can be accessed in e_2 using "`_`", which helps formulate implied loops. Also, "`_`" is automatically replaced by the symbolic value, so this example could also be written as `1..3 => _+4`. Of course, it could also be written as `(1..3)+4`. The parentheses are required because "`..`" has a lower precedence than "`+`", which allows `n..n+10` to produce the "expected" result.

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

```
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. Unlike "`=`", aliases require no declarations and dynamically take upon the right-hand operand's type. Lvalue aliases reduce the need for temporary pointers.

Expressions using `=>` can be nested, and "`__`" can be used to reference a second-level value:

```
gdb> duel ..100 => if (&&/(2.._-1 => __%_)) _
3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

produces the prime numbers up to `100`. The operator `&&/` specifies `andall`. It is a reduction operator that apply a logical conjunction to all its operands. The `||/` operator is similar; it logically-ors its operands.

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, for example,

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

prints the `name` field of the elements in `hash` that have a `scope` greater than `5`. References to "`_`" refer to the left operand's value, which helps eliminate temporaries and simplify expressions; for instance, the example above can be reduced to

```
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.u < 0 || y.u > 100) y
y = { u = -9, v = 0 }
y = { u = 120, v = 0 }

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

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.u < 0 || y.u > 100) x[{j}]
```

The operator `-->` specifies the `dfs` node described in Chapter 4. e_1`-->`e_2 produces the values from the data structure given by e_1 using e_2, which is evaluated under the scope of e_1, 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[[2]]->scope = 2
hash[0]->next[[3]]->scope = 1
```

prints the `scope` fields of the list emanating from `hash[0]`. The symbolic display algorithm automatically prints occurrences of `->`a`->`a as `-->`a`[[2]]`, etc. Specifying `hash[..1024]` would print the `scope` fields for all of the `symbols` in the table.

The example introduced in Section 1.1

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

checks if the list `L` contains 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`. The closely related `L-->next->(value >? next->value)` checks if a list is sorted. Cycles are not detected by `-->` but the special cases of pointing to the head of the list and to "self" can be managed with `L-->(next !=? L)` and `L-->(next !=? _)` respectively.

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 a 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 than 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 `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 last example displays the 7th through the 10th `scope` fields of the 287th entry in the `hash` table. Also, e`[[..`n`]]` returns the first n values from e, and e`[[0]]` returns the first value. It is an error for a selected value not to exist.

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

```
gdb> duel root->(left,right) => #/(_-->(left,right)->key)
#/(root->left ...) = 3
#/(root->right ...) = 1

gdb> duel hash[..1024] => #/_-->next >? 6
#/(hash[119] ...) = 8
```

The first expression counts the elements for each side of the binary tree. The second finds which of the entries in `hash` hold over `6` values. The symbolic result of "`#/`" shows the first counted element followed by "`...`" (this is the actual symbolic output).

The operator 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 a linked list and its 4th and 9th nodes each contain `27` the duplication can be displayed by:

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

In contrast, the expression on page 56

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

displays only the 4th element, and another query like `L-->next->value ==? 27` would be required.

The operator e_1`@`e_2 specifies `stopat`. It produces the values of e_1 until e_2 is non-zero. e_2 is evaluated under e_1's scope, either as e_1`.`e_2 or e_1`->`e_2, when e_1 is a structure or a structure pointer, and as e_1`=>`e_2 otherwise. For example, if `s` is a `char` pointer, `s[0..999]@(_=='\0')` produces `s[0]`, `s[1]`, ... up to but not including the terminating null character. When e_2 is a constant, it is compared to the values of e_1, so the above can be rewritten as `s[0..999]@0`. Also, "e`..`" generates an essentially infinite sequence of integers beginning at e, so `argv[0..]@0` generates the strings in `argv`, an array of string pointers, and `s[0..]@0` generates the characters of `s`. A more complex example is

```
gdb> duel hash[..1024] => (_-->next@(scope < 5))->name
```

which displays the `name` fields in each linked-list of `hash[`i`]` up to an entry with a `scope` less than `5`.

Access to the target's run-time stack frames is produced by `frame(`n`)`, which returns a reference to the nth stack frame; `frame(0)` refers to the procedure frame at the top of stack and `frames_no` contains the total number of frames. Both are reserved words. Frames can be compared to a function pointer to determine if a frame is an activation of that function. They can also be used with "`.`" to evaluate the right-hand operand under that frame's scope, for example,

```
gdb> duel (frame(..frames_no) ==? factorial).n <? 2
```

searchs all the active invocations of `factorial` for a local variable or parameter `n` less than `2`. And finally,

```
gdb> duel (frame(..frames_no) ==? f)[[0]].x
```

returns the value of `x` for the most recent activation of `f`. The parenthesized expression returns all the active `f` frames, of which the first (most recent) is selected and used as the scope for `x`. Because of the usefulness of this case, the special syntax `f.x` accomplishes the same result.

5.2 Differences from C

As discussed in Chapter 3, dynamic scoping rules and the emphasis on expressions make some differences from the C syntax unavoidable. The thorniest issue is user-defined types (`typedef`s.) In *Duel*, user-defined types must be preceded by the keyword "`T`". For example, if `sec` is a user-defined type, then the syntax to declare a variable `x` of that type is "`T sec x`". The `T` keyword is also required in casts and `sizeof`, but a special exception allows `(sec *) x` and `(sec) y` casts without `T`. Because `(t)x` is parsed as a cast operator, the `sizeof` operator for expressions must use parentheses, for example, `sizeof(x)`. Also, expressions like `(printf)("hi")`, which are legal in C, are parsed as casts and produce errors. This compromise stems from C's syntax and the dynamic scoping of *Duel*, combined with the idioms users found most useful.

A related issue is the use of "`;`" before `else`. It is required in C, but must not appear in *Duel*. Similarly, `{x}` is an expression; C requires a semicolon before the closing brace, but *Duel* disallows it; C treats the next token after the closing brace as part of a new statement, but in *Duel* the next token continues the current expression. These differences are tripped over most often in C expressions like

```
if(x) y(); else {z(); g();} f()
```

which in *Duel* must be rewritten as

```
if(x) y() else {z(); g()}; f()
```

These differences stem from *Duel*'s design as an expression-based language. They occur rarely in daily use. Also, *Duel*'s parser "forgives" a semicolon just before `}` and `else` by ignoring it, and parses `]]` as two closing brackets instead of the select operator when such parsing is unambiguous, for example, in `x[y[i]]`.

5.3 Output Manipulation

Duel output can be post-processed with a user-defined program. When activated, output from *Duel* is piped into the user program, and that program's output is displayed. Output to the pipe emits the value and the symbolic value on separate lines to simplify processing. The complete "`duel`" command line is also sent to the pipe. *Duel* ignores input on the command line after "`|>`", the `pipe` operator. This input is passed to the pipe for further interpretation.

This post processing was intended to spark interest in improvements of *Duel*'s output and to help pass it to other tools, but users showed little interest in this feature, partly because it was poorly documented. Currently, only a sample pipe program written in Perl is available. It provides `dec` and `hex` commands to display integral values in decimal or hexadecimal bases and, a `smart` command which displays all output that contains no symbolic values on a single line, for example,

```
gdb> duel 16,17
16
17
gdb> duel 16,17 |> hex smart
0xA 0xB
```

Further improvements to *Duel*'s output, for example, to display matrixes in multi-column output, should be easier in a language like Perl or Icon rather than C.

Chapter 6

Implementation

C is a "bare" language: little information can be extracted from its types and values regarding their validity. Pointers can point "anywhere," including illegal locations, even in a valid target state. Untagged unions, casts that reinterpret any bit sequence, out-of-bounds yet valid array indexing, etc., complicate the implementation of a high-level debugger. While C's complexities are inherent, programmers need to explore invalid target states even in "safe" languages like Pascal, which makes any high-level debugger difficult to implement. For example, it must be possible to display the valid elements of a linked list, even if it contains a cycle or a bad pointer. Only in a few languages, such as ML, can the debugger rely on the language's type safety (Tolmach 1992).

Programmers want to have their cake and eat it too: they want the debugger to pinpoint target-state errors during explorations, for example, to indicate that `p[2]` is an invalid pointer when evaluating `p[..100]->code >? 0`. At the same time, they want the debugger to stay out of their way so they can use suspicious but valid expressions, like `p[-1]->code`, for state explorations.

Duel attempts to address these goals by adhering to the ANSI C standard in the implementation of C idioms, allowing all "undefined behavior" to simulate that of the compiler used to compile *Duel*, while also keeping track of the computation so that errors like bad pointers can be reported in due time. The validity of pointers is verified only when needed. For example, `(int)p[2]` would normally convert and return `p[2]` as an integer even when `p[2]` is a bad pointer. Therefore, *Duel*'s indirection, "`*x`", does not produce an error for a bad pointer `x` until its rvalue is fetched. Symbolic values are constructed on-the-fly, so in evaluating `p[..100]->code+1`, *Duel* reports that `p[2]` is invalid and displays "`p[2]`" as the offending pointer, along its value and the offending operator.

Duel is implemented as an add-on to existing debuggers and is currently interfaced only to `gdb`, although it was not derived from `gdb`. The modified `gdb` contains one new command, `duel`, that parses and evaluates a *Duel* expression and prints the result.

In choosing implementation options, for example, the lexical analyzer or the symbol-table data structure, simpler and easier to modify solutions were chosen over more efficient ones. This choice led to short, flexible code that sacrifices performance. Nevertheless, performance is acceptable. Further improvements are needed only if *Duel* is extended for use in conditional breakpoints and watchpoints as discussed in Chapter 7.

6.1 Parsing

A simple, hand-written lexical analyzer and a `yacc`-based parser are used. A *Duel* expression is compiled into an abstract syntax tree (AST) with nodes corresponding to the primitive operators described in Chapter 4.

As discussed in Chapter 5, dynamic scoping combined with C constructs like `(`*type*`)x` casts results in an ambiguous grammar. Slight modifications to the syntax and the addition of the keyword `T` solves these problems — *Duel*'s syntax is simple to parse. Operator precedence is handled with `yacc`'s precedence rules. The AST is made of dynamically allocated structures implemented in a straight forward way following the description in Chapter 4. The combined parser and lexer are only 750 lines of C.

Little semantic analysis is done at compile time. Type symbols, following `T` and in `(`*cast*`)` are looked up, and the type they represent — not the type name — is entered in the AST. Other symbols are entered in the AST as names without type checkings because of dynamic scoping. Checks also catch several syntactically correct but semantically illegal constructs such as "`x.1`".

6.2 Values and Types

During the evaluation of a *Duel* expression, values are kept in a `value` structure. A value is made up of a type, a value kind, a symbolic value, and an actual value. The value kind is either an lvalue, rvalue, or bvalue. An lvalue is represented as a reference to the target's space where data of the value's type resides. The actual value is the reference value, i.e., a pointer. A bvalue is similar to an lvalue, but refers to a bit field. The actual value is a pointer to the target's structure plus a bit offset and a field length.

An rvalue is a non-referenced value. Its actual value is a value of a primitive C type. *Duel* rvalue's are limited to scalars, so space is allocated for them in a union whose largest element is of type `double`. Therefore, a value is represented by a fixed-size structure, simplifying memory allocation. Currently, *Duel* avoids structure rvalue by modifying structure assignment so that it returns an lvalue and by not supporting calls to C functions that return structures.

The evaluation of a symbol usually returns an lvalue that references the symbol's address. For bit-field symbols, a bvalue is returned. Symbols that reference registers are returned as rvalues with the register's value, which eliminates the need for "regvalue" tag to reference registers at the price of not being able to modify values stored in registers. Registers are a nuisance to manage and are machine dependent, for example, when a `double` value resides in a register pair. Modifying registers is possible with a standard `gdb` command, but under normal debugging sessions, most compilers avoid storing program variables in registers, making the issue moot.

The symbolic value associated with a value is stored as a fixed-size string of at most 120 characters. This implementation avoids the need to manage memory for symbolic values; longer symbolic values are of little value to the user. Symbolic values longer than 117 characters are truncated and followed by "`...`".

Types are stored as a `ctype` structure, which holds atomic type components and points to other `ctype` nodes as needed. For example, a pointer to an array of five integers is represented as a "pointer" node, pointing to an "array of" node that holds the size `5` and a pointer to the "base" node for integer type. This representation is similar to, but simpler than, typical compiler implementations.

Type nodes are most often allocated when a variable is looked up. The type of most results is either a "base" type or is derived from the type of one of the operands, for example, in `x->y`, the result type is taken from the description of field `y` in `x`'s type. `ctype` nodes are never deallocated but might be reused; `ctype` pointers to the same type are not guaranteed to be equal. This approach simplifies the implementation but slows down type checking.

C has a serious type equivalence problem during debugging. `struct` types `a` and `b` are equal if they have the same type name. But name equivalence cannot be established by the debugger when `a` and `b` originate in different modules. The same type name can be used in different modules for different purposes, and untagged structures included via `#include` in several files are often assigned unique names by the compiler — names which differ between modules.

Duel uses *structural equivalence* for all types, including `struct`s. But implementing such comparison is complex and slow due to self-references — a structure can contain references to other structures that reference the first structure, causing a simple recursive type comparison to loop. The current implementation considers two `struct`s equal if their size is equal. This choice seems sufficient since such checks occur only in structure assignments. It should also be checked in parameter passing, but `gdb`'s implementation provides no prototype information.

6.3 Evaluation

Evaluation is implemented by `duel_eval`, which is the actual function corresponding to the abstract function `eval` used to describe the semantics in Chapter 4. `duel_eval`'s code for most of the operators is similar to the pseudo-code that describes their semantics, but many details need to be handled. For example, the code for most C binary operators is

```
bool duel_eval(tnode *n,value *v) {
   bool ok=FALSE ;
   value u, tmp ;

   if(!n) return FALSE ;
   switch(n->node_kind) {
    case OPK_BIN:  /* a+b, compute and hold a, iterate on b, redo a */
      while(n->eval.level==2 || duel_eval(n->kids[0],&n->eval.v1)) {
        n->eval.level=2 ;         /* left side active op in vals[0] */
        while(duel_eval(n->kids[1],&u)) {
           tmp= n->eval.v1 ;      /* copy left val, it is destroyed */
           ok=duel_apply_bin_op(n->op,&tmp,&u,v);
           if(ok) goto done;
        }
        n->eval.level=1 ;   /*left side val no longer valid, re-eval*/
      }
    break ;
    case ... /* other operators */
   }
done:
   if(!ok) n->eval.level=0 ;              /* no other val available */
   return ok ;
}
```

`duel_eval` returns true if it returns a value in its second parameter and false otherwise; there is no actual `NOVALUE`. Temporary values are stored in `n->eval.v1` (inside the AST nodes). `n->eval.level` is the equivalent of `n->state` in the semantics description, but the value `0` is reserved to indicate a non-evaluated node for clean up. The control flow uses fewer `goto`s, but is equivalent to that in the semantics description, except that `duel_apply_bin_op` can return false, indicating no result, for example, for `0 >? 1`. The code is hard to read and manage, because it implements the complex control of generators. Luckily, it is small and independent of the rest of the implementation, for example, from value and type management. `duel_eval` and associated functions are about 400 lines of C.

Implementing functions to correctly evaluate C operators, even in C, is non-trivial; it takes over 1200 lines of C. For example, ANSI C specifies a machine-dependent result *type* of adding an `unsigned int` and a `long`, and does not guarantee that doing all integer arithmetic using `long` and converting the result to the correct type will yield the same result as performing arithmetic in the result type (`gdb` implements arithmetic in this way). A compiler writer may chose a machine-dependent arithmetic implementation, but *Duel* is machine and compiler independent; it evaluates expressions exactly as the compiler that compiled *Duel* would. Doing so requires code like this, for multiplication:

```
void do_op_multiply(value *v1,value *v2,value *r)
{
   ctype *t=find_numeric_result_type(v1,v2,r,"*");
   get_numeric_val(v1,"x*y",t);         /* convert v1 to type t */
   get_numeric_val(v2,"y*x",t);
   r->val_kind=VK_RVALUE ;
   r->ctype=t ;
   duel_set_symb_val(r,"%s*%s",v1,v2);
   switch(t->type_kind) {
   case CTK_INT:
        r->u.rval_int   =v1->u.rval_int   * v2->u.rval_int  ;break;
   case CTK_UINT:
        r->u.rval_uint  =v1->u.rval_uint  * v2->u.rval_uint ;break;
   case CTK_LONG:
        r->u.rval_long  =v1->u.rval_long  * v2->u.rval_long ;break;
   case CTK_ULONG:
        r->u.rval_ulong =v1->u.rval_ulong * v2->u.rval_ulong;break;
   case CTK_FLOAT:
        r->u.rval_float =v1->u.rval_float * v2->u.rval_float;break;
   case CTK_DOUBLE:
        r->u.rval_double=v1->u.rval_double*v2->u.rval_double;break;
   default: duel_assert(0);
   }
}
```

Other operators are more complicated, for example, addition must also handle pointer arithmetic. This code is typical and illustrates several points. First, the function `find_numeric_result_type` computes the result's type based on the types of the operands `v1` and `v2`. This function also sets the symbolic value in `r` using the parameter `"*"` and `v1`'s and `v2`'s symbolic values.

Next, the operands are verified to be numeric, fetched, and type-promoted to `t` by calling `get_numeric_val`. The string `"x*y"` passed to `get_numeric_val` is used only for error messages. Finally, the returned value kind, type, and symbolic values are set, and the actual value is computed, using the compiler's implementation of "*" for each result type.

Dynamic type checking and scoping is managed by `duel_eval` with code as explained above. *Duel* provides no symbol-table management of its own except for aliases and the `with` operand stack. Symbols are looked up as aliases, then in the stack, then by accessing the debugger's symbol table. When errors are detected, information about the current AST node being evaluated is displayed. Symbolic values help users understand the cause of errors: the offending operand's symbolic value is printed, for example, if `z[3]` is an invalid pointer, the expression `5 * z[..100]->code` would produce

```
Error:   5 * z[3]->code
         --^-- illegal address for operand x of 'y*x'
operand ``z[3]->code''  -- type: int
                   -- value: ref @00000008 [ILLEGAL]
```

This error is produced by `get_rvalue` which is called by `get_numeric_val`.

6.4 Performance

As for other very high-level languages, type checking in *Duel* must be done during evaluation, not during compilation. For example, in `(x,y).a`, `x` and `y` can each be a different 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 (Walker and Griswold 1992). 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 symbolic value of an expression is computed at the same time that the expression is evaluated, for example, 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, for example, 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. As a specific example, `1..1000000;`, which produces no output, takes 27 seconds. Removing the code for computing the symbolic value in "`..`" decreased execution time to 12 seconds. Symbol lookup time is also substantial. `0+(..10000)` takes 9 seconds, `i+(..10000)` takes 25 seconds (i's rvalue is fetched 10000 times), `i+0+(..10000)` also takes 9 seconds, and `0+(..10000)+i` takes 47 seconds, since `i` is looked up and fetched 10000 times. While these measurements are incomplete, they are representative of the savings that a better implementation might make. A speed factor of 10 can be achieved by avoiding redundant symbol lookups, value fetches, and symbolic evaluations. Even more improvements could be made with static type checking and compiling ASTs into an intermediate code or even machine code.

6.5 The Debugger Interface

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, as explained above.

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 650 lines of C broken down as follows.

180	comments, directives, help
150	converting between `gdb` and *Duel* types
150	symbol table, registers and frames access
70	accessing target space and target function call
50	converting between `gdb` and *Duel* values
50	`duel` command, miscellaneous

Duel calls functions in the interface module 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 directly. *Duel*'s debugger interface consists of the following functions.

Function	Explanation
`duel_malloc`	allocates debugger memory
`duel_free`	deallocates debugger memory
`duel_get_target_bytes`	copies `n` bytes from a target address
`duel_put_target_bytes`	copies `n` bytes to a target address
`duel_get_target_bitfield`	copies `n` bits from a target bit field
`duel_put_target_bitfield`	copies `n` bits to a target bit field
`duel_target_func_call`	calls a function in the target
`duel_get_target_variable`	returns value and type information for a symbol
`duel_get_frames_no`	returns the number of active stack frames
`duel_get_function_for_frame`	returns a function pointer for a given frame
`duel_get_target_typedef`	returns type information for a typedef symbol
`duel_get_target_struct_tag`	returns type information for a struct tag
`duel_get_target_union_tag`	returns type information for a union tag
`duel_get_target_enum_tag`	returns type information for an enum tag

Except for type and value conversions, most of these functions simply call `gdb` equivalents. Memory allocation in the target does not require a debugger-dependent function — *Duel* locates and calls the target's `malloc`. If `malloc` is not defined, variables cannot be defined and strings cannot be used; `gdb` uses the same method. A trivial implementation of this interface, taking 200 lines, allows *Duel* to run as a stand-alone program capable of complex arithmetic evaluation.

Duel has been ported experimentally to another C debugger (SAS Institute 1992) with relative ease. It was interfaced successfully with `gdb` 4.2 to `gdb` 4.10 under different UNIX systems including SUN, DEC, HP, IBM, and AT&T and under OS/2. The stand-alone version also runs under MS-DOS. Changes in `gdb` versions required only a few modifications to the interface, all because `gdb`'s internal structures changed. An expert familiar with a debugger should be able to adapt *Duel* to run under it in less than a month.

6.6 Discussion

The biggest difficulty in implementing *Duel* was `gdb`. I insisted on implementing *Duel* within an existing debugger, and chose `gdb` because of its wide use. But `gdb` contains over 150,000 lines of code, most of which are poorly documented and have been modified repeatedly to incorporate new functionality, architectures and operation modes (e.g., remote debugging, C++, unusual registers layouts, etc.) *Duel* is implemented in 4,300 lines of C.

I spent months decrypting the interactions between `gdb` modules. Because I was implementing *Duel* as an independent tool, I have not used any of the internal `gdb` functionality such as value and type management. Instead, I crafted code to convert `gdb` information into internal *Duel* structures. It turned out that only a few hundred lines of code were needed, but writing them required a full understanding of `gdb`. It helped to be able to debug `gdb` itself by using another copy of `gdb`. This approach allowed me to debug *Duel* and the *Duel*-`gdb` interface by using both *Duel* and `gdb`.

The experience with *Duel* and `gdb` provides insight into building and designing tools at the complexity level of `gdb`. It is not sufficient to design modules at a functional level; modules should provide independent, general services, with the expectation that they will be used throughout the life cycle of the product in unplanned ways. Different levels of functionality must be kept separate even if combining them for the first release leads to shorter, simpler coding. `gdb` provides few clean interfaces; the function that looks up a local symbol also returns a global symbol if the local isn't found, and there is no function to go directly to the `n`th frame. These capabilities do exist, of course, but they are buried in functions that also provide higher level services, which made the implementation of *Duel* more difficult.

Chapter 7

Extensions and Future Work

Duel is already a complex, operator-rich language, so the addition of "features" that complicate it must be weighted carefully against their use, especially when adding new operators or enhancing existing semantics. Some operators, however, could be simplified by semantic enhancements, for example, the `++` operator is undefined for rvalues, so after `i:=5`, the expression `i++` is illegal. Extending the semantics to permit rvalue aliases would allow idiomatic use like `for(i:=0 ; i<10 ;i++)`.

Duel expressions can be used in conjunction with other debugger operations. The integration of *Duel* with conditional and global breakpoints raises many new semantic and implementation issues. A more general approach to conditional execution, based on *Duel* expressions, emerges from such extensions.

Duel's prime use is in the analysis of error states, often producing a complex expression whose result indicates an error, for example, a linked list with duplicate elements. Using that same expression to execute the program in reverse until the condition becomes false might help automate the search for the bug origin.

Chapter 2 details the problems with existing debuggers, especially the lack of conditional reverse execution or forward reexecution that simulates it. The discussion below concentrates on enhancements to existing facilities, for example, watchpoints, but it applies equally well to reverse execution and the new conditional-execution facilities suggested in Chapter 2.

7.1 Improvements to Duel

Static type-checking can improve the performance of *Duel* as discussed in Chapter 6. It would also eliminate some of the incompatibilities with C and the `T` keyword. Other static checks could lead to improved performance, for example, if the address of a *Duel* variable is never used, it can be allocated in the debugger address space.

Static types and scopes, however, limit the usefulness of the language. It enforces single-type operands and results for all operators. Sensible expressions become illegal, for example, `(x,y).z` for different structure types `x` and `y` both having a field `z` would be illegal because the operands do not have the same type. The alternative is to retain the dynamic types and scopes and to use type and scope inference techniques to provide efficient compilation when possible. Inference techniques can detect subexpressions that produce single-type results, allowing them to be compiled.

Automatic inference isn't perfect. Static typing would evaluate `(1,1.5)+3` by converting the values to `double`. The expression will therefore have a single-type result. In contrast, dynamic typing, even with automatic inference, would be forced to compute a result with two types. It remains to be seen how well an inference algorithm would do in practice, and how often users take advantage of dynamic typing.

As noted in Chapter 3, the implementation of symbolic values is a compromise, for example, `x ==? y` returns `x` and its symbolic value when `x` is equal to `y`; `y`'s symbolic value is lost. Some queries would be more useful or easier to write if `y`'s symbolic value was retained, for example, `a[..10] ==? b[..10]` compares the arrays element-wise; if `a[4]` and `b[6]` are both `3`, the symbolic output is `"a[4]=3"`. A more useful output would be `"a[4]==b[6] = 3"`. The symbolic output for an alias — its name, not the symbolic it references — presents a similar compromise. New operators that provide alternative symbolic values and more control over output warrent investigation.

Several debuggers, for example, `sdb` on UNIX system V, support limited use of regular expressions to resolve names of variables. *Duel* could support a similar feature, for example, if `z` is a structure, then `` z.‘comp_*‘ `` might return the fields of `z` whose names start with `comp_`, making multi-field selection and display easier. Extending such matching to support multiple scopes would allow *Duel* expressions to replace specialized debugger commands, for example, to display the parameters of the currently active function or all global variables declared in a certain module.

Duel functions would allow user-defined generators. Functions are one of the features that all frequent users of *Duel* have asked for. They are most useful in providing "expansion" of data structures, for example, a relatively complex *Duel* expression that returns all the elements of a tree could be defined as a function that is invoked twice to compare the elements of two trees.

In Icon, data structures can be expanded using the "`!`" operator. C provides enough information for *Duel* to automatically expand only some data structures, for example, arrays and pointers to self-referencing structures. The new operator `\x` would expand `x`, so if `head` points to a linked list, `head-->next` could be written simply as `\head`. By allowing users to overload the `\`e operator, e would be expanded by matching its type to a user-function parameter. Although `\x` is just syntactic sugar, it allows users to think of a complex data structure as an expandable unit. Unlike user-written C code, expansion of this form has multiple uses; elements can be selected, displayed, modified, etc.

The limitations of the graph expansion operator that need to be solved are cycle detection and improved control, for example, expanding a tree up to a certain depth. A possible implementation would allow operators like `cycle()` and `depth()` to be used from within the expansion expression, for example,

```
root-->(if(depth(i:=(left,right)) < 10) i)
```

would expand a binary tree to a maximum depth of 10. Cycle detection, however, costs time, so special "fast" search operators, that could be used when the user is sure that no cycles exist might be useful. Other searching algorithms, for example, breadth-first search, can be added as new operators.

The limitations of the selection operator e_1`[[`e_2`]]` also require more work. It is impossible to reference the "last" element of e_1 and retrieving elements with decreasing indexes is inefficient, because the generation of e_1 is restarted for each element.

Finally, several operators have not been implemented, including pair-wise comparison of generators like `==/`, summation operators like `+/`, and C modified assignments like `+=`. Other unimplemented features include modification of bit fields, initialization of declared variables, declaration of new types, a step-specified range operator (e.g., `1..100:+2`), and structure and array constants. User experience indicates these extensions are useful, but not essential, although they might be more important if *Duel* were used in conditional and global breakpoints.

7.2 Conditional Breakpoints

A conditional breakpoint is a breakpoint that occurs only under specified conditions. For example, when line 48 is reached, execution stops only if the condition `x==0` is true. Conditional breakpoints are the simplest form of "automatic debugging." A typical scenario occurs when a function `f(x)` behaves incorrectly for a negative parameter `x`. Using a breakpoint with the condition `x<0` at `f`'s entry stops execution when the first negative `x` is passed to `f`. An unconditional breakpoint would stop at each call to `f`, forcing the programmer to check `x`'s value manually and to continue execution, possibly hundreds of times, until a negative `x` is occurs.

In most debuggers, the expression used in conditional breakpoints is a source expression, limiting their use. For example, it is impossible to stop when an array contains a negative element or a function `f(x)` is called from function `g` with a negative `x`. Conditional breakpoints are an advanced debugging technique; when used, non-trivial conditions are likely to be called for — conditions that cannot be expressed with a source expression but are easy to formulate with *Duel*. Examples of *Duel*-conditions likely to be used with conditional breakpoints include

```
frame(0)==f && frame(1)==g && f.x<0
&&/(frames_no>9, frame(..10)==h)
#/root-->next <? 3
||/(y[..100]<0)
```

The first condition stops if `f` is called from `g` and `x` in `f` is negative. The second condition stops if the function `h` is called recursively more than ten times. The third example stops if the linked list pointed by `root` has less than three elements, and the last example checks for a negative element in array `y`.

Conditional breakpoint performance is critical. Waiting a few seconds for the evaluation of a *Duel* expression that was entered interactively is acceptable, but if the same expression is evaluated hundreds of times in a breakpoint before execution is halted, an unacceptable delay of over ten minutes could occur. *Duel*'s performance must be improved, as discussed in the previous section.

Some *Duel* conditions are easier to express by returning a non-Boolean result. For example, `x[..100] ==? 0` intuitively stops when `x` contains a zero element, but it returns zero when the condition is met and nothing otherwise. To return a Boolean result, it must be rewritten as `||/(x[..100]==0)` or `#/(x[..100] ==? 0)>0`. User experience is needed to best decide how conditions should be specified and if special syntax or semantics should be added, for example, to specify that a condition is met if more than one value is returned.

In some cases, it is useful to declare variables that are "local" to the breakpoint but whose values are retained across repeated evaluations of the condition. For example, `int count=5 ;; --count>0` could be used as a condition to stop execution after the breakpoint is reached five times. Another example is

```
int oldx[100] ; oldx[i:=..100]=x[i] ;; ||/(oldx[j:=..100] != x[j])
```

which stops when it detects that array `x` was modified. The "`;;`" operator separates the initialization from the condition. When the breakpoint is reached the first time, both the initialization and the condition are evaluated. Thereafter, only the condition is evaluated. Alternatively, the initialization is evaluated immediately.

Allowing variables in breakpoints creates many complexities that merit further investigation (Crawford *et al.* 1992). Examples include visibility, interactive access to the values, and the storage of multi-value results, for example, the current values of a linked list.

7.3 Watchpoints and Global Breakpoints

Conditional breakpoints are limited to a single program location. Programmers often require that a condition be checked in multiple locations or even everywhere. For example, a user may learn that a variable is modified gratuitously, for example, overwritten by a bad pointer. A global conditional breakpoint that checks its condition everywhere, would locate that modification. Global breakpoints conceptually single-step the program and check the condition at each step.

Watchpoints, or traces, add syntactic sugar. A watchpoint is a global breakpoint that stops when the value of an expression changes. For example, if `x` is `5`, a watchpoint on `x` would be equivalent to a global breakpoint on `x!=5`.

Both watchpoints and global breakpoints are rarely used. Linton (1990) suggests these are powerful idioms and notes their poor performance, but also indicates that users have not asked for improvements. The reason for lack of use might be that they are of little use in their current form. As with conditional breakpoints, these advanced features are likely to require non-trivial conditions that cannot be specified with a source expression, but could be expressed in *Duel*. Examples include a hash table being over 90% full, a matrix containing a zero column, assignment to an array element that was non-zero, and a tree containing a cycle.

The issues regarding conditional breakpoints apply here, too, but some are aggravated. Performance is crucial because the condition is typically checked tens of thousands of times and variables "local" to the conditions might be required for many common conditions.

Scoping issues present a serious problem. As the program executes, the scope where the condition is checked changes. Current debuggers essentially ignore the scoping issue. Borland uses dynamic scoping; a watchpoint for `x` can change from a global to a local variable. `gdb` deletes a watchpoint as soon as the variables it references go out of scope, which makes it impossible, for example, to set a watchpoint on a local variable that goes in and out of scope.

Further research is required concerning scope, and especially the ability to refer to global or module-wide variables from the condition. Variables could be specified based on a location from which they are visible and an active stack frame if they are local variables to a recursive function, for example, variable `x` visible from line `56` in module `eval.c` using the 3rd activation of this function. Syntax to express such information might be tedious to type, so mouse interface issues should be considered.

Duel expressions already support partial checking of the current execution location, for example, `frame(0) == strcpy` is true if the program is in the function `strcpy`. Programmers can use such expressions to limit global breakpoints to only specific program locations, for example, a breakpoint that stops when a linked list is modified outside the scope of functions that are known to modify it. Semantics should therefore be added to allow pinpointing the current execution location. I have been unable to design a good syntax that solves both this and the previous scoping problem satisfactory. Such syntax should allow references to specific location or locations, to a function entry or exit, and to a module or source file.

Finally, watchpoints for a *Duel* expression are more complicated than watchpoints for a simple, single-valued expression. One option is to execute the condition when it is first entered and to save the resulting values. Alternatively, a `changed()` builtin operator could be provided, for example,

```
changed(x[50..99]) || ||/x[..50]==0
```

would stop as soon as the upper half of array `x` is changed or an element of the lower half becomes zero. The best semantics for `changed` are unclear. One possible implementation would collect the operands at the first stop and thereafter compare the operands to those previously collected.

There are several different ways to implement fast watchpoints for simple variables (Wahbe 1992). It is possible to analyze *Duel* watchpoints and to derive a set of simple variables whose change is a precondition for the *Duel* condition to be true. Using fast, simple watchpoints on these variables could eliminate many needless evaluations of the *Duel* condition therefore speeding up program execution.

7.4 Execution-Control Conditions

If *Duel* was enhanced so that it could check the current execution location, it would be possible to write global breakpoints that are semantically equivalent to breakpoints on source locations, for example,

```
if($$ == scope("eval.c",45)) stop
($$ ==? scope("eval.c",32)).if(x==5) stop
```

The first expression is an unconditional breakpoint at line 45 in module `eval.c`; the second is a conditional breakpoint at line 32 in `eval.c`, depending on `x==5`, `x` being the visible `x` at that line. The `$$` operator refers to the "current location." Such global breakpoints, especially if written in an idiomatic fashion, could be optimized by the debugger into regular and more efficient breakpoints. These semantics provide an alternative to the complex and often incoherent debugger commands that deal with different kinds of breakpoints and watchpoints. The target always appears to be single-stepped, and a set of programmer-specified conditions is checked, or evaluated, after each interesting source location. Each *execution-control condition*, or ECC, is written in an extended *Duel*-like language. ECCs could provide the sole control over execution and would be optimized by the debugger to provide efficient program execution where possible. An empty ECC would execute the program to the end, an ECC consisting solely of `stop` would single-step the target, etc.

Modern debuggers often provide single-key or mouse-click actions to insert breakpoints or single step the program. The same actions can be used to create ECCs, freeing the programmer from writing relatively long code for simple actions. Debugging commands like `next` and `return` could also be specified in terms of ECCs.

ECCs are a significant change from current debuggers and debugging methods. They require a new debugger centered around them, an ECC optimizer and efficient interpreter, and a new user interface. There are many syntactic and semantic issues that need to be solved, and an actual implementation and user experience with this new environment is required. Most of the ideas in the previous sections could be incorporated or modified for use with ECCs.

In contrast to ECCs, Crawford *et al.* (1992) suggest that execution control be achieved by invoking specialized code that single steps the program from within user-specified breakpoint code. For example, in order to have a watchpoint in effect only in a certain function, a breakpoint is set at the entry to the function and the breakpoint code single-steps and checks the watchpoint condition until the function returns.

They note, however, that invoking single stepping from within breakpoints poses thorny semantic problems. The reason for allowing such semantics is to provide efficient execution, for example, single stepping only during the execution of the checked function. Using ECCs and assigning the task of efficient execution to the debugger (via optimizations) provide a more coherent debugging model in which programmers need not be concerned with low-level mechanisms used to implement their conditions.

7.5 Other Extensions

Programmers add assertions to programs to verify that certain conditions hold true at certain locations. In C, this is accomplished by using the `assert` macro. With other languages, like Eiffel (Meyer 1992), assertions are built in. In either case, however, assertions are written in the source language. Non-trivial assertions require complex coding and are therefore often avoided. By providing *Duel*-assertions or a similar language extension for writing assertions, programmers might use assertions more often. Assertions are important both for debugging and for documentation. Traditionally, predicate calculus has been used to describe assertions with existential and universal qualifiers (Gries 1981). The use of generators in *Duel* provides a comparable facility.

Many debuggers provide interactive exploration of data structures, which sometimes provide an alternative to *Duel*. Certain ideas can be borrowed from *Duel* to enhance interactive exploration. For example, after an array of structures is displayed, the user could use interactive mouse actions to limit the displayed fields or the range of values being displayed, similar to simple *Duel* expressions. Expansions of linked lists and other simple data structures could also be provided.

An even better approach is to integrate interactive exploration and the use of

Duel expressions. A *Duel* expression may be "entered" in a specific data cell being displayed. That cell would provide the scope for the expression and its value would be accessible through "`_`". The results of the evaluation of *Duel* expressions would be displayed graphically, to guide further interactive exploration. For example, a complex query to locate the largest element of array `x`, then use that to index array `y` of link-list pointers, and retrieve the tail of that linked list, could be entered interactively as follows: first, a *Duel* query outputs the largest element of `x`, which is displayed graphically. The *Duel* expression `y[_]` is evaluated at this location, producing the element of `y` indexed by the displated element. Next, the *Duel* expression `_-->next` produces the elements of the linked list. Finally, the user might browse to the tail of the list using mouse actions.

Chapter 8

Conclusions

Since the introduction of symbolic source-level debugging in the late 70's, the process by which programmers debug their code has seen little change. Many programmers continue steadfast in their use of print statements as a debugging tool, ignoring debuggers altogether. Those who use debuggers suffer from several problems common to most debuggers.

Debuggers often frustrate their users. Details of the target's state, which the user knows "are there," like the value at a tail of a linked list, cannot be accessed; high-level facilities like watchpoints cannot be used because they are too inefficient, and reverse execution, even a few steps back, is often impossible. Attempts to work around these problems often result in unwieldily debugging methodologies, which ultimately cause users to stop using debuggers.

This thesis has touched only the tip of the iceberg. *Duel* attempts to improve the state querying facility, but much more work is required. The importance of good debugging facilities will continue to grow as software systems become ever larger and more complicated. Users should be educated to pay as much attention to programming tools as they give to performance benchmarks. Researchers should realize that advancing software development and debugging is as important and "exciting" as improving, for example, disk-drive performance.

8.1 User Experience

Duel is public-domain software; a previous version as reported in (Golan and Hanson 1993) has been available for several months.[1] That version of *Duel* produced a list of all commands entered in a file and I have asked people to mail these files back to me for analysis.

Despite being technically complex to install and use, the release was a moderate success. In about three months, over a hundred people installed and used *Duel*. Feedback from these users helped me to understand the difficulties of using *Duel*, to modify several aspects of *Duel*, and to substantiate the importance of such a tool. Nevertheless, it is important to realize that these comments, however encouraging, are not the product of a controlled experiment with results that can be assured statistically.

SAS Institute has incorporated *Duel* into an experimental version of their C debugger (SAS Institute 1992), and intends to support it in the next release. They have reported no special difficulties in interfacing with the *Duel* code. Overall, response was enthusiastic; some of the more vivid comments I received were:

> "I like *Duel*. I was getting really sick of typing `*ml->next->next->next->next` ad naseum,"

and

> "*Duel* seems to have lots of varied functionality packed into terse operators, rather like Perl . . . I really hope that many of your ideas are picked up by debugger-writers. If so, you've done a great service to software developers."

Stephen Harrison seems to have summed up the opinions of most users:

> "Just occasionally something turns up that's good enough to make you wonder how you ever managed without it. *Duel* is in this category."

[1] *Duel* is available for anonymous ftp from `ftp.cs.princeton.edu:/duel`; users must have the source code for `gdb`, also available for ftp at the same site.

Several significant bugs were also discovered by users, indicating that *Duel* has been actively used. A serious problem with this kind of "experiment" is that people avoid sending negative comments. I had to encourage such comments in order to gain a more objective picture. Probably the biggest complaint was the syntax, for example,

> "Minor nit; I think you'd agree that the syntax leaves a little to be desired—it took me an hour or two to commit the common idioms to memory."

A common reason for not using *Duel* was the use of another debugger and the unwillingness to use `gdb`, even with *Duel*. The only graphical interface for `gdb`, called `xxgdb`, pales in comparison to some of the interfaces available for other debuggers. Still, no other debugger whose source code is available is also as widely used. Within two weeks of its release, *Duel* has been installed and tested on a variety of systems including UNIX and OS/2 systems by SUN, DEC, IBM, HP, and AT&T. Designing *Duel* to be portable paid off. The few portability problems encountered were often environmental, such as the lack of an ANSI C compiler, although in several cases I modified the code to improve its portability.

The importance of *Duel*'s succinct expressions remained unclear, but as examples below show, at least some succinct capabilities are important. Several users made specific comment on its importance, while others found the syntax tedious to learn, and reported that they used the help facility often. The existence of over 50 examples of common use in the manual was widely appreciated, and writers of similar software would do well to devote time and space for more examples. At least some terse operators were used by everyone, for example, the "`..`" operator.

The idea of using a different language to express queries about the target state seems to be supported by everyone who tried *Duel*. Most users accepted it subconsciously, simply finding that *Duel* fills the bill. Others noted it, as one user said:

> "Programming languages are not designed to explicitly make queries about the state of a program. In addition, for the most part, these sort of queries are independent of the source language."

There were many requests for enhancements. Several people asked for *Duel* "procedures" and C++ support. Others wanted improved formatting of numerical results, for example, the current version always prints unsigned integers in hexadecimal. The idea of using *Duel* expressions everywhere in `gdb` — especially in conditional breakpoints — excited several users. One user who wanted to debug optimized code needed syntax like `$pc` to access registers. With my help, he modified *Duel* to support it and his code was added to the latest release. Finally, two small companies who develop debugging tools showed interest in integrating *Duel* into their code.

Several people wanted to postprocess the output from *Duel*, for example, to display data points as a graph. To that end, I added support for post processing of the output using a UNIX filter as discussed in Section 5.3. This facility, however, was optional and poorly documented, and I have received little feedback about its use. Instead, one user reported extending *Duel*/`gdb` to work with a `REXX` interpreter, and another modified the `emacs-gdb` integration software to post process *Duel*'s output.

Among the most common problems were the need to use the keyword `T` with type definitions. In response I have modified the parser so that `(T type)` and `(T type*)` casts, the cases used most often, are accepted without requiring the `T` keyword. User complaints also led to initializing all declared variables to zero and to some improvements in parsing, for example, `x[y[1]]` is accepted even though `]]` is the selection operator. These changes indicate that careful design does not eliminate the need for hands-on experience.

An analysis of the command files verified most of my expectations. Most expressions are simple and use only a few operators. One commonly used idioms is `a,b,c` simply to display the values of several scalars with their symbolic names. In contrast, `gdb` requires three `print` statements and the symbolic names are lost. Both the `..` and `-->` operators are used heavily, often only to print a range of values. The most clearly misunderstood feature was the use of "`_`" with "`=>`", for example, several users used expressions of the form "*e* `=> _`", which are identical in all respects to just *e*.

As expected, users entered increasingly complex expressions by interactively building on previously entered expressions. It is unclear, however, how much of this use should be attributed to typical exploratory use versus the learning curve of *Duel*.

Users did make complex queries. The following examples were all entered by users who debugged real programs. The systems being debugged included a Lisp interpreter and an SQL manager. Each line is a different example.

```
 sb->messages-->next[[29]]->n_fields
*(struct sqlvar_struct *)frame(2).$i0
this->right_tree->tree->nodes.things[..15]
#/hMetaTree->root-->(oldestChild,youngerSibling)
mastart[..10][..10] => if(_.tisOri != _.tisMod) _
(infx_sqldas->i_sqlda.sqlvar[..41]).(if (sqlind[0] != -1) _)
((T CONS*)list)-->((T CONS*)cdr)->(if (car->type==8) frob(car));
((INFX_TABLE_HDR *)curr_inf_ptr)->infx_sqldas->k_sqlda.sqlvar[..2]
*(short *)(((INFX_SQLDAS*)0x1aabcc)->k_sqlda.sqlvar[2,3,6].sqldata)
pstaSimHead-->pstaSimNext->(tisMod,pstaTrainNextStationPrev->tisMod)
```

8.2 Duel

The problem that *Duel* addresses did not "exist" beforehand. It was through attempts at creating more advanced debugging tools that I have realized that the current state querying facilities in many debuggers are inadequate. If *Duel* has accomplished nothing else, it has at least articulated the need for improved state querying facilities.

Common wisdom suggests that data created by C code is best manipulated by C code. Therefore, simple C-expressions are used in debuggers for target exploration. Such use masks the problems; existing limitations are often blamed on the lack of a complete C interpreter in the debugger. Yet some necessary operations cannot be expressed in C, for example, searching the target's active stack for certain function activations. The limitations inherent in debugging a language with itself apply to any language; it is certainly not a problem of C or C debuggers per se.

The design of *Duel* proved much harder than I imagined. What started as a short-hand notation with emphasis on output formatting ended as a full-featured language closely resembling C. In retrospect, it should have been clear that *Duel*'s design would be difficult. Because the problem did not "exist," *Duel* had to "prove" the problem's existence. Specifically, *Duel* had to be "clearly better" than slapping a C interpreter into the debugger, although showing that *Duel* is an improvement is a subjective matter that cannot be fully resolved.

Hoare (1983) suggests that

> "The language designer should be familiar with many alternative features designed by others, and should have excellent judgment in choosing the best and rejecting any that are mutually inconsistent ... One thing he should not do is to include untried ideas of his own."

It seems that *Duel* ignores these suggestions in every conceivable way. But extending C with multi-value semantics and adding a whole slew of new operators, all in a specialized language for a new task, seems impossible under Hoare's guidance. Clearly, some language designers must break the rules, or new ideas will never emerge. Going through two cycles of syntax and semantic changes has improved the language, and user experience confirms that the current design is at least suitable for its intended domain. Nevertheless, user experience indeed indicated several problems with the "new ideas," most of which I was able to fix.

Duel is oriented towards the debugging of C. The ideas could be adapted to Pascal, Modula, etc., but the resulting language will be significantly different. *Duel* breaks away from traditional attempts to make the debugger language-independent. Debuggers, like compilers, are and should be language dependent, although a multi-language debugger, like a multi-language compiler, is a fine idea. All too often it seems that rather than providing several debuggers for different languages and needs, designers have chosen to provide a single, lowest common denominator debugger.

Duel is an example of a specially-designed language for debugging. It uses generators for the underlying semantics and works in a "hostile" environment, where neither garbage collection nor full data-structure expansion is possible. It provides symbolic evaluation to produce meaningful output automatically. Experience with *Duel* indicates that such a debugging tool is welcomed by users.

Duel's implementation is also non-traditional. Rather than implement *Duel* as an independent tool, or as a "toy" debugger, I was inclined from the start to implement *Duel* such that it can be used by real users "in the field" in order to test its effectiveness. Much good research is lost or ignored because no serious implementation that others can use is provided; just as often flaws in a research idea are not exposed until a serious implementation is attempted.

Implementing *Duel* to interoperate with an existing debugger whose source code is available leaves only a few options, for example, `gdb` and `dbx`. `gdb` was an obvious choice, given its widespread use and portability. I was so certain of that choice that I did not check the alternatives seriously. As it turned out, `gdb` is very complicated. Much time and effort went into understanding `gdb`'s internals and building "around" it to avoid having *Duel* depend on `gdb`. Whether the choice of `gdb` was wise remains unclear, although it certainly paid off in the experience gained by having many users.

This thesis presents several possible extensions to *Duel* and to the design and implementation of other debugging facilities. Some are contradictory, like execution-control code (ECC) and the insertion of breakpoints directly into source code. Only experience can tell which is better. Forward reexecution, which is a simpler alternative to reverse execution, coupled with conditional-breakpoints that use *Duel* expressions, seems like the most promising extension.

Chapter 7 discusses the use of *Duel* expressions in conditional and global breakpoints. While the discussion details various problems that such extensions raise, it should be remembered that existing facilities use only a source expression, and even the use of *Duel* "as is" would still be an improvement.

Some might suggest that the use of high-level languages and object-oriented techniques will confine most bugs to simple objects, reducing the need for languages like *Duel*. The current trend, however, is to use low-level facilities in a language like C++ to provide high-level objects; bugs might indeed involve only a few objects, but the details of these objects and their implementation will need to be examined to explain these bugs. Since general-purpose objects are likely to be implemented with complex data structures, facilities for state query will remain invaluable for exposing bugs.

8.3 Improving the Programming Environment

A debugger is but one tool in the programmer's toolbox. The interactions between the compiler, editor, and the debugger affect the usability of the debugger. The definition of the programming language, and its "spirit," as it is captured in common implementations, effect the way debuggers operate and the kind of bugs that debuggers help to locate. The discussion below reflects on these issues based on the experience with *Duel*.

Integrated development environments (IDEs) are often touted as a solution to interaction problems between programming tools. But even though some IDEs are now commonly used, many programmers prefer stand-alone tools, especially in large projects. No single IDE answers the needs of all projects; mixing and matching smaller tools — the "UNIX approach" — is likely to continue to dominate large projects.

Mixing and matching requires standard interfacing. UNIX provides just such an interface with pipes and standard i/o redirection. A similar, commonly available, interface among programming tools does not exist. `gdb` has no programming interface which complicated the implementation of *Duel*. This problem is not new or unique to *Duel* — many tools have been developed in conjunction with or on top of debuggers using ad hoc methods, for example, `dbxtool` (Adams and Muchnick 1986).

Interaction with editors is also problematic. Few editors provide an interface that allows debuggers to display source in the editor with the current location highlighted. `emacs` (Stallman 1981) is one of the few general-purpose editors that provides an interface to the debugger, but rather than allowing the debugger to control the editor, the implementation is reversed: the editor attempts to control the debugger using the standard debugger commands, which are intended for a human interface. This approach is limited, for example, `emacs` cannot display multiple breakpoints in a special color.

Further research is needed in the design and specification of editors, compilers, and debuggers that can interface to and control each other. For example, it should be possible for an external tool to specify which lines to highlight in the editor and to indicate errors or breakpoints. Compilers should allow interactive error recovery, so an error could be directly reflected in the editor, fixed, and compilation continued. Debuggers should provide breakpoints and execution-location management and display through the editor.

Many committees have standardized languages, disk controllers, etc. Standardizing the debugger command-language and the interaction between common programming tools could reap similar benefits, for example, most debugging information is kept in a proprietary, often ad hoc format, making the design of third-party debuggers hard or impossible. DWARF (UNIX International 1993) is an on-going attempt to standardized at least that aspect of the compiler-debugger interaction.

Tools to browse source code and to display it at various levels of detail can prevent programming errors and help in detecting and fixing errors when they occur. Traditionally, browsers are stand-alone, special-purpose tools, but they too can and should be integrated with the editor. Such integration would allow programmers to display the declaration of a variable with a mouse click within the editor, for example. The benefits of each such "look up" during editing are relatively small, so leaving the editor to run a special browser each time is impractical. The combined benefits of many "look ups" is significant, but can only be realized from within the editor.

The design of the programming language plays an important part in the editor's ability to display source-code in intelligent ways. It is unfortunate that C, and especially C++, are complicated and hard to analyze, making the design of a combined editor-browser much harder. Recent extensions to C++ exacerbate these problems.

While language committees spend much time arguing minor details that might affect portability, little time is spent considering the standard's implications on other programming tools. For example, ANSI C ignores the problem of deciding when two types declared in different modules are equivalent, which forces *Duel* to use an ad hoc technique to compare types. Whether a closing brace "`}`" that ends a loop is a "statement" on which the user can place a breakpoint and whether `continue` would reach that statement are unspecified, so different debuggers handle these situations differently. Pseudo code to interact with the debugger is also not defined, for example, a pragma like `#break` to force a breakpoint at that location.

The C language is also poorly suited for debugging. It specifies no run-time checks, has a weak type-system, and its pointer semantics often cause memory overwrites. It therefore seems that C and C++, unless heavily modified, cannot address the above issues, although a safe subset of C++ has been designed by Ellis and Detlefs (1993).

Modula-3 does address most of these issues, but it is incompatible with C style, syntax and semantics — perhaps the main reason why it has been largely ignored. New languages that address the above issues while trying to stay compatible to C's "spirit" are worth serious consideration.

Appendix A

Duel Reference Manual

A.1 Precedence

Table 1 summarizes all of *Duel*'s operators in decreasing order of precedence. All of the C operators are implemented in *Duel* with the same precedence. Most of the *Duel* operators use the same precedence level as in C, e.g., the `>?` operator has the same precedence as `>`. Several *Duel* operators, most notably `#` and `@`, use the highest precedence level, even though a lower precedence level would have been more appropriate. This choice forces users to use parenthesis more often, but avoids the confusion possible with the addition of more precedence levels.

A.2 Operators

Each of the operators in Duel is described in details in the following sections, including special side effects, evaluation order and caveats.

Binary C operators

The binary 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]
```

<table>
<tr><th>Associativity</th><th>Operators</th></tr>
<tr><td>left</td><td>{x} (x) x[y] x->y x.y x(y) x-->y x[[y]] x#y x@y</td></tr>
<tr><td>right</td><td>#/x -x *x &x !x ~x ++x --x x++ x-- (cast) x
frame(x) sizeof x</td></tr>
<tr><td>left</td><td>x/y x*y x%y</td></tr>
<tr><td>left</td><td>x-y x+y</td></tr>
<tr><td>left</td><td>x<<y x>>y</td></tr>
<tr><td>none</td><td>x..y ..x x..</td></tr>
<tr><td>left</td><td>x<y x>y x<=y x>=y x<?y x>?y x<=?y x>=?y x>?y</td></tr>
<tr><td>left</td><td>x==y x!=y x==?y x!=?y</td></tr>
<tr><td>left</td><td>x&y</td></tr>
<tr><td>left</td><td>x^y</td></tr>
<tr><td>left</td><td>x|y</td></tr>
<tr><td>left</td><td>x&&y &&/x</td></tr>
<tr><td>left</td><td>x||y ||/x</td></tr>
<tr><td>right</td><td>x? y:z</td></tr>
<tr><td>right</td><td>x:=y x=y x+=y, ...</td></tr>
<tr><td>left</td><td>x,y</td></tr>
<tr><td>right</td><td>x=>y</td></tr>
<tr><td>right</td><td>if(x) y else z while(x)y for(u;v;w) x</td></tr>
<tr><td>left</td><td>x;y</td></tr>
</table>

Table 1: *Duel* operators in decreasing precedence

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 reevaluated 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.

Unary C operators

The unary C operators

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

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. Casts to a user defined type must begin with `T`, e.g., `(T val*) x`. See variable declaration for more details.

`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 with single-value right-hand 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)` continues 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

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

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.

Basic Generator: `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 `x[[y]]` or `x@y` operators. The precedence of "`..`" is between "`<`" and "+" 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 pair 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. 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 operators work as in C for a symbol `y`. If `y` is an expression, it is evaluated under the scope of `x`, e.g., `z.(a+b)` is the same as `z.a+z.b`, if `a` and `b` are field of `x` (if 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 "`_`" in `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 the symbol `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, e.g., `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 after the variable goes out of scope. The special command `dl clear` deletes all the aliases, and `dl alias` shows 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` (including "`_`".)

`x@y`

The expression `x@y` produces the values of `x` until `x.y` is non-zero. This allows expressions like "`for(i=0 ; x[i].code!= -1 && i<100 ; i++) x[i]`" to be written as `x[..100]@(code==-1)`. The evaluation of `x` stops 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 operators 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 is non-zero. As 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 `y`th result of `x`. If `y` returns multiple values, each selects 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 `0`th element). Be careful with side effects in `x` — its evaluation is restarted for decreasing `y` values.

`frame(n)` `frames_no` `func.x`

`frame(n)` returns a reference to the `n`th 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 fourth frame is a call to `myfunc`, and in scope resolution, e.g., `frame(3).x` returns the local variable `x` of the fourth 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`. "`(frames(..frames_no)==?f)[[0]].x`" which access `x` for the top most invocation of `f` can be written simply as `f.x`.

References

Adams, Evan and Steven S. Muchnick. 1986 (July).
DbxTool: A window-based symbolic debugger for Sun workstations.
Software—Practice & Experience, 16(7):653–669.

Agrawal, H., R. A. Demillo, and E. H. Spafford. 1993 (June).
Debugging with dynamic slicing and backtracking.
Software—Practice & Experience, 23(6):589–616.

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. 1988.
The AWK Programming Language.
Reading, MA: Addison Wesley.

American National Standard Institute, Inc. 1990.
American National Standards for Information Systems, Programming Language C ANSI X3.159–1989.
New York.

Appel, Andrew W. and Andrew P. Tolmach. 1990 (June).
Debugging Standard ML without reverse engineering.
In *ACM Conference on LISP and Functional Programming*, pages 1–12, Nice, France.

Aral, Ziya, Ilya Gertner, and Greg Schaffer. 1989 (May).
Efficient debugging primitives for multiprocessors.
Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices, 24(5):87–95.

Balzer, Robert M. 1969.
EXDAMS—EXtendable Debugging and Monitoring System.
In *AFIPS Proceedings Spring Joint Computer Conference*, pages 567–580, Arlington, VA. AFIPS Press.

Bates, Peter and Jack C. Wileden. 1983 (August).
An approach to high-level debugging of distributed systems.
Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices, 18(8):107–111.

Bates, Peter. 1989 (January).
Debugging heterogeneous distributed systems using event-based models of behavior.
Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, 24(1):11–22.

Beander, Bert. 1983 (August).
`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.

Bovey, J. D. 1987 (September).
A debugger for a graphical workstation.
Software—Practice & Experience, 17(9):647—662.

Brooks, Gary, Gilbert J. Hansen, and Steve Simmons. 1992 (July).
A new approach to debugging optimized code.
ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 27(7):1–11.

Brooks, Frederick P. 1987 (April).
No silver bullet: Essence and accidents of software engineering.
Computer, 20(4):10–18.

Bruegge, Bernd and Peter Hibbard. 1983 (August).
Generalized path expressions: A high level debugging mechanism.
Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices, 18(8):34–44.

Budd, Timothy A. 1982 (March).
An implementation of generators in C.
Journal of Computer Languages, 7(2):69–82.

Cargill, T. A. and B. N. Locanthi. 1987 (Oct).
Cheap hardware support for software debugging and profiling.
Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices, 22(10):82–83.

Cargill, Thomas A. 1983 (December).
The Blit debugger.
Journal of Systems and Software, 3(4):277–284.

———. 1986 (January).
The feel of Pi.
In *Proceedings of the Winter USENIX Technical Conference*, pages 62–71, Denver, CO.

Coutant, Deborah S., S. Meloy, and M. Ruscetta. 1988 (July).
DOC: A practical approach to source-level debugging of globally optimized code.
Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 23(7):125–134.

Crawford, Richard H., Ronald A. Olsson, W. Wilson Ho, and Christopher E. Wee. 1992 (April).
Semantic issues in the design of languages for debugging.
In *Proceedings of the International Conference on Computer Languages*, pages 252–261, Oakland, CA.

Ellis, John R. and David L. Detlefs. 1993 (June).
Safe, efficient garbage collection for C++.
Research Report 102, System Research Center, Digital Equipment Corp., Palo Alto, CA.

Feldman, Stuart I. and Channing B. Brown. 1989 (January).
IGOR: A system for program debugging via reversible execution.
Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, 24(1):112–123.

Fraser, Christopher W. and David R. Hanson. 1991 (October).
A retargetable compiler for ANSI C.
SIGPLAN Notices, 26(10):29–43.

Fritzson, Peter. 1983 (August).
A systematic approach to advanced debugging through incremental compilation.
Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices, 18(8):130–139.

Gimpel Software. 1986.
PC-Lint 2.00 Manual.
Collegeville, PA.

Golan, Michael and David R. Hanson. 1993 (January).
DUEL — a very high-level debugging language.
In *Proceedings of the Winter USENIX Technical Conference*, pages 107–117, San Diego, CA.

Gramlich, Wayne C. 1983 (August).
Debugging methodology (session summary).
Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices, 18(8):1–3.

Gries, David. 1981.
The Science of Programming.
New York: Springer-Verlag.

Griswold, Ralph E. and Madge T. Griswold. 1986.
The Implementation of the Icon Programming Language.
Princeton, NJ: Princeton University Press.

———. 1990.
The Icon Programming Language. second edition.
Englewood Cliffs, NJ: Prentice Hall.

Griswold, Ralph E., David R. Hanson, and John T. Korb. 1981 (April).
Generators in Icon.
ACM Transactions on Programming Languages and Systems, 3(2):144–161.

Griswold, Ralph E. 1982 (October).
The evaluation of expressions in Icon.
ACM Transactions on Programming Languages and Systems, 4(4):563–584.

Hanson, David R. and Ralph E. Griswold. 1978 (May).
The SL5 procedure mechanism.
Communications of the ACM, 21(5):392–400.

Henderson, Peter and James H. Morris. 1976 (January).
A lazy evaluator.
In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 95–103, Atlanta, GA.

Henry, Robert, Kenneth M. Whaley, and Bruce Forstall. 1990 (June).
The University of Washington illustrating compiler.
Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 25(6):223–233.

Hoare, Charles A. R. 1983.
Hints on programming language design.
In Horowitz, Ellis, editor, *Programming Languages: A Grand Tour*, pages 31–40.
Rockville, MD: Computer Science Press.
Reprinted from *Conference Record of the ACM Symposium on Principles of Programming Languages*, Boston, Oct. 1973.

Hseush, Wenwey and Gail E. Kaiser. 1989 (January).
Data path debugging: Data-oriented debugging for a concurrent programming language.
Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, 24(1):236–247.

Intel Corporation. 1986 (April).
80386 data sheet.
Santa Clara, CA.

Iverson, Kenneth E. 1962.
A Programming Language.
New York, NY: John Wiley & Sons.

Johnson, Mark Scott. 1977a (October).
The design of a high-level, language independent symbolic debugging system.
In *Proceedings of the ACM Annual Conference*, pages 315–322, Seattle, WA.

Johnson, Stephen C. 1977b (December).
Lint, a C program checker.
Technical Report 65, Computing Science Research Center, AT&T Bell Laboratories, Murray Hill, NJ.

Johnson, Mark Scott. 1978 (August).
The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment.
PhD thesis, The University of British Columbia.

———. 1982 (April).
Some requirements for architectural support of software debugging.
SIGPLAN Notices, 17(4):140–148.

Kessler, Peter B. 1990 (June).
Fast breakpoints: Design and implementation.
Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 25(6):78–84.

Knuth, Donald E. 1973.
The Art of Computer Programming: Volume 1, Fundamental Algorithms. second edition.
Reading, MA: Addison Wesley.

Linton, Mark A. 1990 (June).
The evolution of Dbx.
In *Proceedings of the Summer USENIX Technical Conference*, pages 211–220, Anaheim, CA.

Lopriore, Lanfranco. 1988 (May).
A user interface specification for a program debugging and measuring environment.
Software—Practice & Experience, 19(5):437–460.

Matanzano, J. F. and S. R. Bourne. 1977 (January).
A tutorial introduction to ADB, UNIX Programmer's Manual, 7th Ed, Vol. 2A.
Bell Laboratories.

Mellor-Crummey, J. M. and T. J. LeBlanc. 1989 (May).
A software instruction counter.
Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices, 24(5):78–85.

Meyer, Bertrand. 1992.
Eiffel: The Language.
London: Prentice Hall International.

Miller, Barton and Thomas LeBlanc, editors. 1989 (January).
Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, volume 24.

Miller, Barton and Charles McDowell, editors. 1991 (December).
Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, volume 26.

Myers, Brad Allan. 1980 (June).
Displaying data structures for interactive debugging.
Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

O'Bagy, Janalee and Ralph E. Griswold. 1987 (July).
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.

Olsson, Ronald A., Richard H. Crawford, W. Wilson Ho, and Christopher E. Wee. 1991 (May).
Sequential debugging at a high level of abstraction.
IEEE Software, 8(3):27–35.

Olsson, Ronald A., Richard H. Crawford, and W. Wilson Ho. 1990 (June).
Dalek: A GNU, improved programmable debugger.
In *Proceedings of the Summer USENIX Technical Conference*, pages 221–231, Anaheim, CA.

———. 1991 (February).
A dataflow approach to event-based debugging.
Software—Practice & Experience, 21(2):209–229.

Powell, Michael L. and Mark A. Linton. 1983 (August).
A database model of debugging.
Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices, 18(8):67–71.

Ramsey, Norman. 1993 (January).
A Retargetable Debugger.
PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ.

Ross, Graham. 1987 (January).
Integral C — a practical environment for C programming.
SIGPLAN Notices, 22(1):42–48.

Russell, Mark. 1992.
`ups`.
`ups` is a graphical, source-level debugger for C. It is available for anonymous `ftp` from host `unix.hensa.ac.uk` (129.12.21.7), in `/misc/unix/ups/ups-2.45.tar.Z`. The author's email address is `mtr@ukc.ac.uk`.

SAS Institute Inc. 1992 (June).
SAS/C Development System User's Guide, Volume 2: Debugger, Utilities, Assembler, Version 6.0.
Cary, NC.

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

Stallman, Richard M. 1981 (June).
EMACS: The extensible, customizable self-documenting display editor.
In *Proceedings ACM Symposium on Text Manipulation*, pages 147–156, Portland, OR.

Steffen, Joseph L. 1984 (April).
Experience with a portable debugging tool.
Software—Practice & Experience, 14(4):323–334.

Tolmach, Andrew P. 1992 (October).
Debugging Standard ML.
PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ.

UNIX International. 1993 (July).
DWARF Debugging Information Format.
Parsippany, NJ.

Wahbe, Robert. 1992 (September).
Efficient data breakpoints.
Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices, 27(9):200–212.

Walker, Kenneth and Ralph E. Griswold. 1992 (August).
An optimizing compiler for the Icon programming language.
Software—Practice & Experience, 22(8):637–657.

Wall, Larry and Randal L. Schwartz. 1990.
Programming Perl.
Sebastopol, CA: O'Reilly and Associates.