Implementing Nickle

Keith Packard
Hewlett-Packard Cambridge Research Laboratory
keithp@keithp.com

Abstract

The Nickle programming language is a modern imperative/functional programming language featuring (among other things): C-like syntax; arbitrary precision integer and rational arithmetic; settable-precision floating point arithmetic; a strong static type system featuring subtypes (that also allows flexible types, permitting run-time type checking); automatic storage management; first-class functions; first-class continuations and exceptions; first-class threads with concurrency control; and a simple but usable module system. Nickle is implemented as a interactive compiler that generates code for a run-time byte-code interpreter. The runtime support for such a project is non-trivial: much of the 25,000 lines of C and 1,500 lines of Nickle in the current implementation is devoted to runtime issues.

Some of the interesting elements of the runtime system will be discussed, including the byte code interpreter, storage representations of data, the C-friendly garbage collector, environments, continuations, thread support, and arbitrary-precision arithmetic. The efficiency and effectiveness of the current system will be discussed, including performance and performance bottlenecks, as well as the potential for other implementations with different engineering tradeoffs.

1 Introduction

The design of Nickle started with the need for a system for arbitrary precision numeric computation. At the time, there were two common ways of performing such computations – building a library and calling it from C, or using Lisp. Neither option provides the graceful infix syntax, and the C option requires a lot of additional bookkeeping on the part of the programmer for memory management.

The language 'ec' (for extended-precision C) was designed along with some basic arbitrary precision integer routines. This was implemented by a compiler which generated code for a byte-coded virtual machine. The compiler was separate from the runtime system, much like Java compilers today. Storage management was done with reference counting, the lack of structured data types made that quite practical.

A separate project involved the development of an interactive calculator language 'ic' for quick numeric work. Early versions of that language evaluated double precision floating point expressions directly inside the parser. That was later modified to build data structures that an interpreter would walk for evaluation. This permitted the addition of stored programs.

The 'ec' language was eventually discarded, and the numeric routines folded into 'ic' so that this interactive language could be used for arbitrary precision numeric computations. Ic grew slowly, adding support for rational arithmetic, strings, files and arrays. With the addition of arrays, the reference counted storage manager was replaced with a mark-sweep garbage collecting allocator taken from the Kalypso lisp interpreter. Ic was a pure polymorphic language, no type declarations are available and there is no type checking.

The data structure interpreter which restricted language developments was replaced with a bytecode compiler and interpreter, support for threading was added and the language name was changed from 'ic' to 'nick'. Threading was added mostly because it was possible, given the byte code interpreter, it also turned out to be useful in implementing a primitive debugger. Primitive variable data typing was added, along with runtime type checking. Structured data types were added, but not fully type checked. Continuations were developed and exposed using the traditional C 'setjmp' and 'longjmp' functions.

Years later, a paper about the language was submitted to the Usenix technical conference. In preparation for this paper, several longstanding nagging issues with nick were addressed. Static type checking was added, the machine-specific floating point code was replaced with a software implementation that provides arbitrary precision. Namespaces provided some separation between interfaces and implementations. Structured exceptions and language support for assured execution across non-local control flow changes address the integration of threading, locking and continuations. As the language had changed in many fundamental ways, the name was changed again to 'nickle'.

2 Nickle Implementation Overview

There are several easily distinguished internal areas within the Nickle implementation:

- Parsing and lexing. Parsing the Nickle language stretches the limits of an LALR parser.
- Bytecode generation. Explicit parse trees are converted to bytecodes. Typechecking requires interactions with the lexer to resolve typedefs and namespaces.
- Bytecode interpretation. The Nickle bytecode language is tightly coupled with the underlying implementation.

- Storage management. A portable mark-sweep garbage collection combined with some C conventions makes storage management within the implementation simple and robust.
- Numeric computation. Classic algorithms along with some new ideas work together in Nickle's numeric system.
- Threading and continuations. Continuations are used throughout the implementation to solve flow of control issues.
- Performance analysis. Nickle provides performance measuring tools that capture statement level timing.

3 Parsing and Lexing

Nickle uses a lexer built with 'lex' and a yacc grammar. The combination of these two tools has dramatically simplified the maintenance and development of the language.

The Nickle parser builds complete parse trees which are saved internally to report messages and errors – instead of saving the program text, Nickle can redisplay any portion of the program directly from the parse trees. The pretty printer output must be carefully generated to ensure it will reproduce the same program when fed back into the lexer. This is currently a manual process; changes to the language must be reflected by changes in the pretty printer.

To retain C-style typedefs, and to permit similar usage for name space names, the lexer must be able to distinguish between various kinds of identifiers. The parser places names into the symbol table, along with their storage class and type. The parser doesn't allocate storage; that operation is left to the bytecode compiler. This means that the parser is responsible for identifying symbols currently in scope.

Before the addition of typedefs and namespaces, the parser passed raw symbol names to the compiler, which made symbol table management and storage allocation a single operation. This conflicted with the desired syntax for the language and the available LALR parser.

4 Compiling to Bytecodes

The compiler walks the parse tree built by the parser to generate bytecodes. Separate blocks of instructions are allocated for each function or top-level statement.

Storage is allocated for each new symbol, and a few unusual scoping rules are applied to prune out names which aren't in scope. One such case is that initialization of static variables cannot use function arguments or local variables. These appear in the static scope of the initializers, but aren't available during execution. Nested functions require the use of a static link for stack and static variables, the depth of which is computed at compile time.

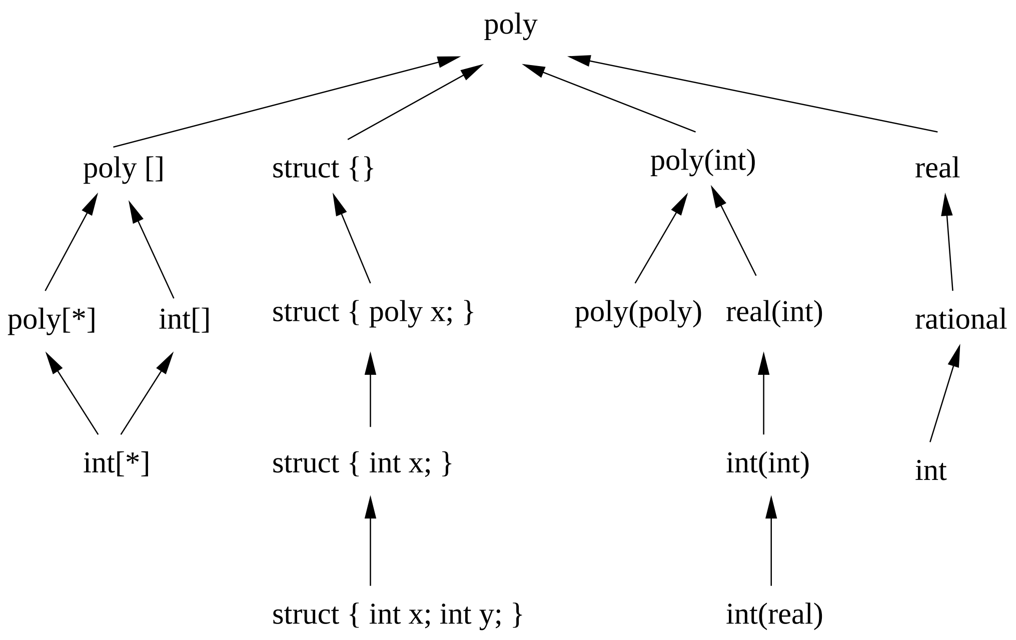

Figure 1: Subtype Relationships

Much of the compilers efforts are centered on type checking. Each operation and variable use are statically type checked. Because the underlying virtual machine is fully polymorphic and performs full run-time type checking, the compiler static type checking is strictly for the benefit of the developer. Type-checking is somewhat ad-hoc in the current implementation, the intent is to require exact type matches, except where expressions use polymorphic types. Two types "match" if either is a subtype of the other, a few examples of subtype relationships are shown in Figure 1.

The type checking rules were recently relaxed to permit subtype to supertype conversions to be left to the runtime system to check. This change allowed the specification of functions like 'max' that could operate over all of the numeric types. Some of this will not be necessary once parametric polymorphism is added to the language, but the 'pow' function (and the identical ** operator) will still need this relaxed rule. 'Pow' returns integers whenever the left operand is an integer and the right hand is a positive integer, but the type system doesn't have a specification for positive integers, hence 'pow' must be specified to return rational numbers for arbitrary int/int operands.

The compiler doesn't currently perform any kind of optimization; the generated bytecodes reflect the program structure directly. Simple optimizations like constant folding would provide some improvement in performance. More sophisticated operations like common subexpression elimination would impact the developers ability to debug applications, and would need to be optional. Such optimizations should be relatively straightforward, given the availability of the full parse tree at compile time. Because the parse tree is used to display the context of errors, a separate tree would be needed for any tree transforming optimizations.

5 Execution in Nickle

The Nickle virtual machine provides an accumulator, a stack of values and a separate list of activation frames. The stack of values is used strictly for intermediate computation, no persistent values are stored there. Local variables and function arguments are stored in the frame; static variables are stored in a block referenced from the frame and globals are stored directly in their symbol table entries.

The combination of an evaluation stack and an accumulator matches C semantics quite well where every expression has a value which may, or may not, be used in further computations. The 'ec' language used a strict stack machine which required a significant amount of bookkeeping when managing function return values and chained assignments. The simple addition of an accumulator eliminate all of that difficulty.

5.1 Program Data Storage

Local variables and arguments are stored in the frame so that continuations can easily capture their storage locations, rather than just their current values. The semantics of static allocation requires that new storage be allocated each time the enclosing function definition is evaluated.

The evaluation of the function definition creates a closure containing storage for static variables declared within that function, a reference to the enclosing static frame of the function and a pointer to the executable code. When the function is called, a frame is built containing a pointer to that storage for the static variables, a pointer to the enclosing static frame and new storage for dynamic variables declared within the function. This frame is linked into the thread execution frame context and the instruction pointer in that context is set to the first instruction in the function.

5.2 Virtual Machine Instructions

The compiler generates "bytecodes", which are simple instructions containing an opcode and up to two parameters. There are currently 58 opcodes, that number varies quite a bit as the structure of the runtime system changes. Many of the opcodes are special purpose operations that manipulate virtual machine state which is specific to the Nickle language, like twixt statements, exception handling and thread creation. There are twelve opcodes used to manipulate storage, with separate opcodes for structures, arrays and structure references.

These are all designed to reduce the conditional code needed within the interpreter for each operation.

Each instruction also contains a pointer back to the parse tree for the statement in which it occurs, this permits the run-time system to report errors relative to the source code, rather than the instruction stream. It would be more efficient to separately store ranges of opcodes related to the same statement, however this implementation is quite a bit simpler.

5.3 Virtual Machine State

All execution occurs in the context of a thread. Threads are one of the primitive data types in Nickle so that Nickle applications can manipulate them directly. As shown in Figure 2, the execution context within a thread contains:

- A continuation containing:
  - The accumulator and the value stack.
  - An instruction pointer along with a pointer to the block of instructions containing the pointer.
  - A pointer to the current frame.
  - A list of current caught exceptions.
  - A list of current twixt blocks.
- Any currently executing long jump.
- Some scheduling state.

5.4 Instruction Restart and Completion

Execution of any opcode can be interrupted by certain signals or an internal exception. This is implemented by instruction restart; the state of the virtual machine must always be updated atomically after the instruction is committed to being completed. Operations which affect global machine state force the current instruction to complete so that any side effects happen only once.

Other signals don't interrupt the current instruction, instead they wait for the current instruction to complete before affecting the machine state. This was necessary, especially for timer and IO signals, to ensure that computation would continue to make progress in the face of slow instruction execution and frequent signal interruptions.

5.5 Breakpoints

The current debugger has relatively powerful inspection capabilities, as it permits access to the full language in the context of the current statement. However, there is no support for suspending execution except when an exception is raised and not handled. Simple breakpoints need to be added to the virtual machine and then exposed to the debugger so that more sophisticated debugging can be performed.

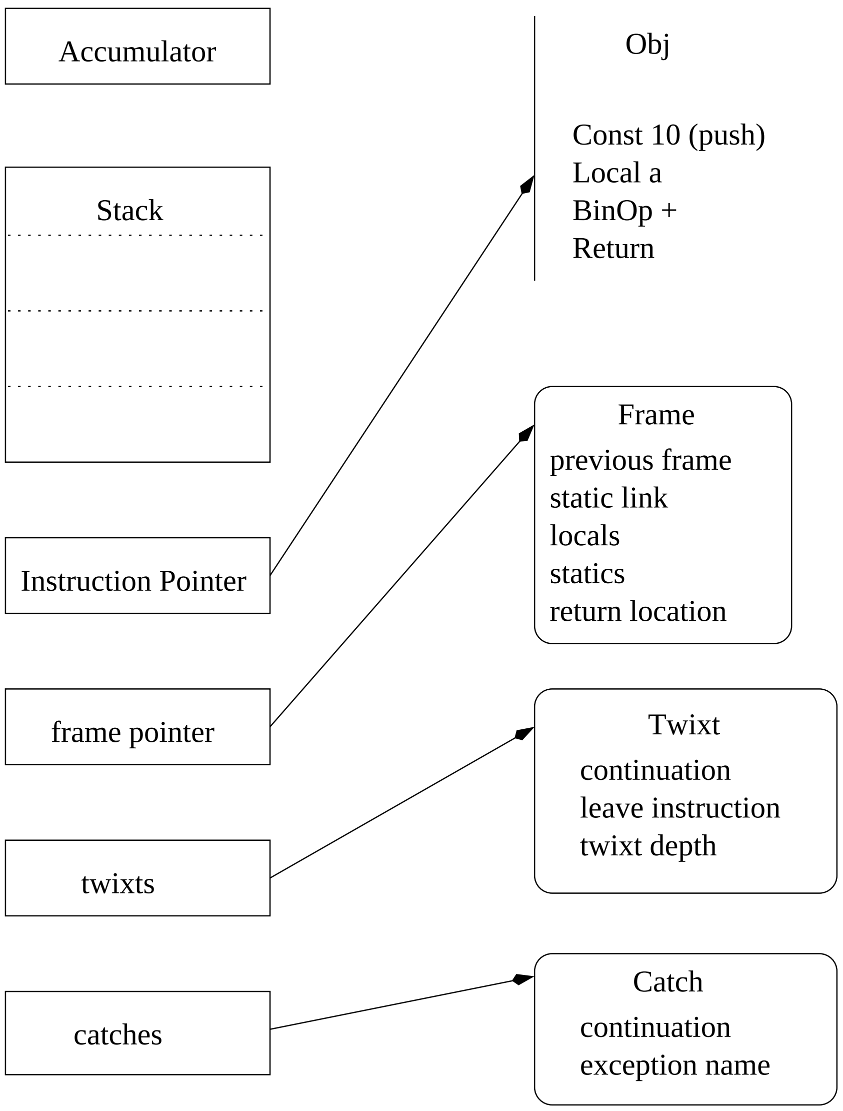

Figure 2: Execution Context

6 Storage Management

One of the design goals of this family of languages has always been automatic storage management. Early languages used reference counting, but with the addition of composite (and possibly recursive) data types, a garbage collector was strongly desired. Fortunately, another language implementation, Kalypso, provided a portable, C-friendly, mark-sweep garbage collector.

To be portable, the garbage collector required that all data be explicitly rooted; the collector couldn't walk the C stack or hunt through machine registers for data references. The Kalypso implementation was constantly plagued by missing references, these would be harmless until the garbage collector was run at the wrong time.

6.1 Garbage Collection in C

When integrating the Kalypso garbage collector, a conservative automatic reference mechanism was designed. Newly allocated memory is placed in a special memory stack when returned from the allocator, this stack is manipulated by the C routines in a simple way – as all new values are placed on the stack as they are allocated, each routine need only reset the stack to it's previous height and insert any values that routine returns.

Routines which fail to participate in this convention don't cause any harm, the values passing through them are left on the stack and the stack is cleaned by routines higher in the call chain. This particular mechanism is effective enough that the entire runtime system now uses garbage collected memory and has largely avoided all storage related problems.

A stop and copy collector would probably provide some significant performance improvements, but it would eliminate the flexible C interface as stop and copy collectors must rewrite all references to every object in the system. Because data outside of that explicitly allocated isn't examined by the collector, there is no way to rewrite references that may be contained in registers, the C stack or unknown global variables.

6.2 GC Data Structures

Memory is allocated in power-of-2 sized buckets, from 8 to 32K bytes, above that, separate malloc chunks are used for each allocation. A single bit of overhead is required for each small allocation to store mark state. Each allocated object must contain a pointer to a structure containing a function to call when scanning the memory system for referenced data, this pointer is used within Nickle for values to also mark the type of value referenced, and to point at the functions which manipulate data of that type. This means that every allocation has four bytes of overhead, making even a simple 32-bit integer consume 8 bytes of memory. Until recently, the values also contained explicit tag values, making the lowly int consume 16 bytes (12 bytes of value, padded to a power of two bucket).

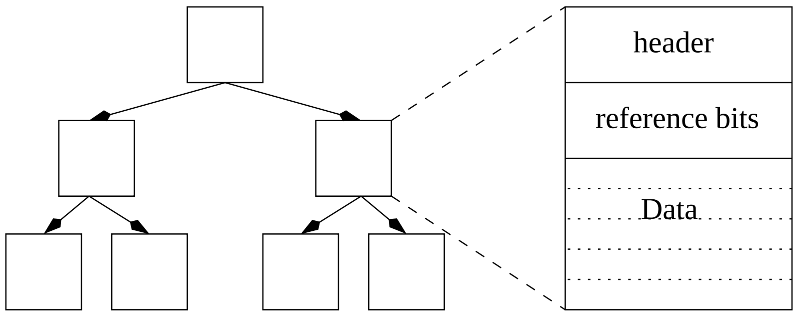

Figure 3: Memory Allocation Data Structure

Chunks of identical sized allocates are grouped together. Each chunk, along with chunks for the large allocations, is placed in an AVL tree as seen in Figure 3 so that the mark operation can locate the chunk based solely on the address of the object within the chunk. An AVL tree was chosen because it automatically balances the tree, even given the typical linearly increasing addresses returned from the underlying system allocator used to allocate the chunks.

7 Numeric Algorithms

As the original 'ec' language was designed for arbitrary precision computations, the Nickle implementation has tried to incorporate reasonable implementations of algorithms for large numbers. The basic natural arithmetic routines are implemented using 64 bit objects, which significantly improves performance over 32 bits.

7.1 Integer Multiplication

Natural multiplies are handled either with the classing "grade school" approach, or, when dealing with number larger than 6400 bits, the Karatsuba algorithm is used. A future addition might include an FFT-based multiplier for really large numbers. Karatsuba multiplies reduce the total cost of computation using the following relation:

$$\begin{aligned}
xy &= (x_1 b + x_0)(y_1 b + y_0) \\
&= b^2 x_1 y_1 + b(x_1 y_0 + x_0 y_1) + x_0 y_0 \\
&= b^2 x_1 y_1 + b(x_1 y_0 + x_0 y_1 + x_1 y_1 + x_0 y_0) + x_0 y_0 - b x_1 y_1 - b x_0 y_0 \\
&= (b^2 - b) x_1 y_1 + b(x_1 + x_0)(y_0 + y_1) + (1 - b) x_0 y_0
\end{aligned}$$

Select b so that the larger of x or y is split into two roughly equal pieces, and recurse to compute the sub factors. This reduces the algorithm from $O(n^2)$

to $O(n^{lg3})$. Because of the larger constant factor, smaller numbers are more efficiently multiplied with the classic algorithm, hence the recursion terminates when one of the terms becomes smaller than the limit described above (6400 bits).

An algorithm with similar performance improvement for division exists, but has not been implemented for Nickle.

7.2 Integer GCD

Computing the greatest common divisor (GCD) of two integers is an important part of any implementation of rational arithmetic; the obvious representation for rational numbers provides many equivalent representations for the same value, it is best to always convert numbers to a canonical representation, done by dividing numerator and denominator by their GCD.

Nickle uses Weber's accelerated integer GCD algorithm, which is nominally $O(n^2)$, just like the classic binary GCD algorithm, except that it does many fewer multi-precision computations making it significantly faster in practice. This algorithm alternates a k-ary reduction with the usual GCD modulus step; k-ary reduction is used when the terms are of similar size, while the modulus is used otherwise.

For operands u and v, the k-ary reduction replaces u with $nv - du$ where n and d are 32-bit values chosen to yield at least 64 zero bits in the result which can then be stripped off. This reduction can introduce spurious factors which are eliminated at the end of the GCD computation by taking the result, g, and recomputing the GCD against the original two inputs: $gcd(u, gcd(v, g))$. As g is generally much smaller than u or v, this computation is usually quite fast.

7.3 Repeating Decimals

One of the interesting problems when dealing with rational numbers is how to display them accurately. As any rational number can be represented by a repeating expansion in any base, Nickle provides a mechanism for computing this representation, presenting the result in three parts:

$$\frac{num}{den} = int.fixed\{repeat\} \tag{1}$$

The length of the *fixed* portion of the representation is calculated by

```
fixed = 0;
while ((g = gcd (den,base)) != 1)
        fixed++;
        den /= g;
```

The length of the *repeat* portion is somewhat more difficult to calculate, and beyond the comprehension of this author who wrote the code long ago and failed to comment it sufficiently.

7.4 Arbitrary Precision Floating Point Numbers

One of the undesirable limitations of the 'nick' language was its use of machine-specific floating point numbers for imprecise computation. More than the limited precision of the mantissa, the inability to convert arbitrary integer or rational values to an imprecise equivalent made them fit very poorly into the Nickle value hierarchy. Several alternatives were investigated to provide some kind of imprecise values which could hold values of arbitrary magnitude

One possible representation is to bound the error of the value on either end with an arbitrary precision rational number. This interval arithmetic has many desirable properties, not the least of which is a very tight and accurate error bound after each computation. It is, however, very expensive in practice. Long computations with rational numbers usually yield numbers with large terms, as floating point computations often involve numeric approximations, the speed of the resulting implementation would be unacceptable in many cases.

Instead, the classing 'floating point' representation is used, this time with arbitrary precision mantissa and exponents. This permits very high precision computations (given suitably precise inputs), and an arbitrary range of magnitudes. Providing more than 56 bits of mantissa isn't generally necessary in the physical sciences, but can be useful for some calculations so Nickle uses a default precision of 256 bits. The extra computation necessary isn't generally significant, and the reduction in cumulative error can be handy at times.

The precision of any operation is affected by the precision of its inputs – multiply a value precise to 256 bits with one of 1024 bits results in a value precise to only 256 bits. Addition and subtraction are a bit different; the precision of the result depends on the precision of the inputs along with their relative magnitude.

Given the underlying implementation of natural numbers, there aren't any special algorithms for dealing with floating point numbers. The imprecise math functions are provided by a library (written in Nickle) which implements sqrt, cbrt, exp, log, log10, log2, sin, cos, tan, atan, asin, acos and pow using a variety of algorithms which can produce values of arbitrary precision.

The floating point representation requires that any conversion from precise numbers be accompanied by a precision argument; there is a default precision, but it would be nice to develop some better convention or mechanism to determine the necessary precision.

8 Threading and Continuations

The switch from a tree-walking interpreter to a byte code interpreter provided some of the most interesting capabilities in Nickle. Because the execution context could be encapsulated in a single data structure, it was easy to permit more than one to exist at a time and to cycle through each in turn, this provides very fine-grain multi-threading, albeit capable of using only a single processor.

Being able to duplicate this context permitted the implementation of continuations in a scheme style – the internal virtual machine state is separate from the program visible values, things like local or static variables can be shared among multiple continuations while each holds its own value stack, accumulator and program counter.

While threading is probably one of the least used features of Nickle by applications, it is necessary for the implementation of a Nickle-based debugger, the debugging code can run in parallel with the target code and interact directly.

Continuations contain a subset of the execution context of the thread, the accumulator and value stack, a frame pointer and program counter, the list of handled exceptions, and the list of currently active twixt blocks. This commonality should be exposed in the implementation by making them use the same data structure, but having threads reference their data through a pointer would cause some performance issues. It might be possible for each to share a common data structure, perhaps at some point that change can be investigated more closely.

8.1 Threading Effects on Nickle

Aside from the relatively trivial implementation of threading within the virtual machine, the effect on the interface of Nickle with the operating system is a bit more complicated. Nickle places all files in non-blocking mode and has each file deliver SIGIO when new data arrive. This allows Nickle to continue executing unblocked threads while others are pending on file descriptors.

The addition of threading to the language has had a much more significant impact on the design of the language, Nickle provides an interesting mixture of structured exception handling, continuations, structured mutual exclusion execution and threading.

9 Twixt, Exceptions and Non-Local Jumps

While programmatic use of twixt and exception handling are usually quite distict, within the interpreter they share many similarities. Both twixt and exception handlers involve non-local flow of control. Twixt ensures that various blocks of code are executed during a non-local control transfer while exception handlers serve as a target for dynamically computed control transfers.

The twixt statement provides a structured mechanism for identifying regions of code protected by various mutual exclusion operations. One of the interesting semantic requirements is that the entry portion of the twixt be executed along any path into the twixt, and the exit portion be executed along any path out of the twixt.

Exception handlers provide dynamically scoped functions that are invoked when exceptions are raised. This means that called functions are generally unaware of the exception handler and will not know the target of an exception.

Both exception handlers and twixt statements place additional state in the thread continuation. As see in Figure 4, each twixt block or exception catch

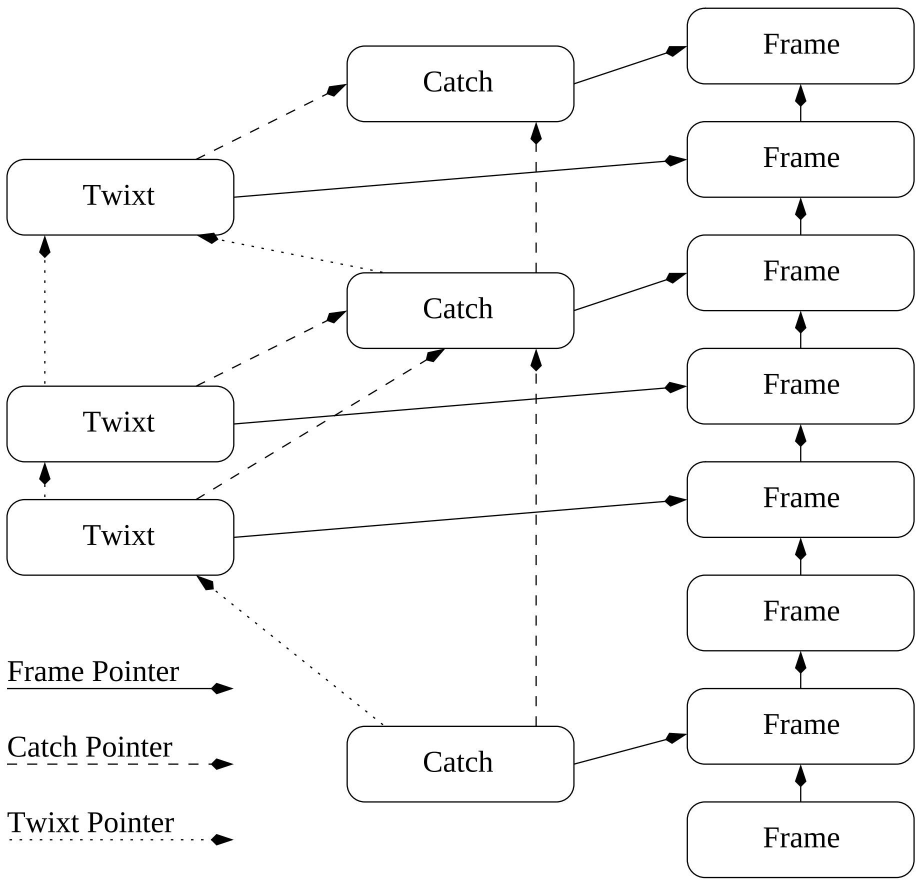

Figure 4: Twixt blocks, Exception Handlers and Function Call Frames

block captures a continuation so that the interpreter can restart execution within their context.

9.1 Twixt

The twixt record contains an additional pointer so that either the enter or leave block may be executed as necessary. The syntax of the language ensures that the continuations for these two blocks are identical except for the instruction pointer, so the interpreter avoids creating two separate continuations by keeping a separate instruction pointer for the leave block. On exit from the twixt statement, the twixt record is removed from the list.

When a thread jumps into a continuation, the transition is mostly just a matter of copying state from the continuation into the thread. However, the two lists of Twixt records must be reconciled so that the appropriate application invariants are true.

Nickle does this by finding the first common twixt block along the two chains and then building a temporary data structure associated with the thread called a Jump record which holds pointers into the two twixt lists to guide the incremental transition from one location to the other as shown in Figure 5. To enter each twixt block, the thread state is set from the Twixt saved state. On exit from the twixt block, the interpreter checks for the presence of the Jump record; if present, it passes control to the next twixt block or on to the target of the continuation if no more twixt blocks need to be handled.

9.2 Exceptions

Each exception handler prepends a catch block to the thread execution context, the catch block contains a continuation to return to the exception handler and the name of the exception. On exit from the try statement, the catch block is removed. When an exception is raise, the interpreter walks up the list of catch blocks looking for a handler. If one is found, control is passed to the handlers continuation. A tiny stub of code at that target passes control to the handler function. This control transfer must check for the presence of twixt blocks as described above.

9.3 Effect of Twixt and Exceptions on Other Code

For non-local jumps caused by break, continue and return statements, Nickle automatically builds a continuation if necessary to ensure that intervening twixt blocks are executed appropriately. This is statically decidable because all of the possible twixt blocks are statically visible from the source statement and the compiler inserts appropriate Unwind instructions which count the number of nesting catch/twixt blocks.

The continuation data structure is shared with threads, twixts and catches. Continuations contain the instruction pointer, frame pointer, stack, catches and twixts. Threads contain additional scheduling state. Twixts contain additional

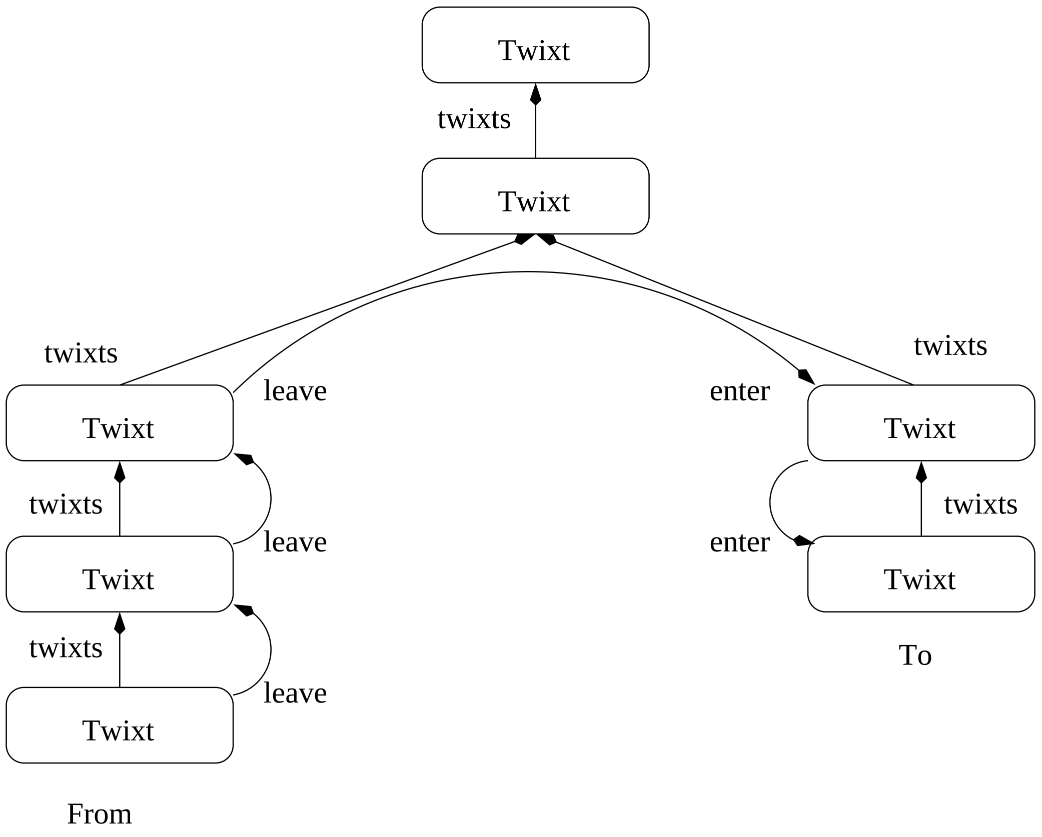

Figure 5: Flow of Control During Continuation Transfer

instruction pointers that point to the start of the enter and leave blocks. Catches are labeled with the exception they catch. Continuations are directly embedded in these other structures to reduce memory allocator overhead and pointer dereferencing in the interpreter.

10 Performance Analysis

To help refine algorithms to spot obvious performance problems, Nickle provides statement level profiling information. This is done by enabling a 10ms periodic timer signal with setitimer. At each timer tick, the current program counter is used to find the associated statement. Two counters in each statement represent the ticks consumed with the program counter within the statement and the ticks consumed with the program counter within some function called from the statement. These values are displayed by the pretty printer.

The Nickle compiler generally uses tail-call optimizations to limit the frame depth due to recursion, but this eliminates statements using tail calls from the cumulative performance data. To help this, the compiler can be instructed not to generate tail call instructions when profiling.

11 Conclusions

The current Nickle implementation is a result of a long and leisurely developement process spanning two decades. Much of the code has been refactored or reimplemented several times leading to areas which reflect a high degree of polish. Some of the code has languished, largely because it works well enough and hence has no impact on correctness or functionality.

Because development has occurred sporatically over such a long time, there are areas of the code which are no longer well understood, and areas which reflect design methodologies of an earlier era. One obvious problem in many areas is the lack of sufficient documentation on the underlying methods used in the implementation. An overall design document has been sorely needed for some time.

The gradual redefinition of the language has left its mark in various parts of the code. There are several data structures which seem roughly bolted together as the language semantics migrate past their original design. The lack of a development schedule has allowed many such instances to be fixed, resulting in a gradual improvement in the overall integration of the system.

Nickle has been the subject of a wide ranging search for ideas in program language development, numeric algorithm development and language implementation. The current language brings significant power to the design and development of software while still being usable as an interactive environment. The implementation attempts to capture resposible engineering practices in straightforward ways while still providing respectable performance and not impeding the desired advancement of the language itself.