Direct Abstraction

Matthew Kaes

ECE 460L

Contents

Abstract:

Direct Abstraction is a combination of several computer science fields including development driven interfaces, cross-platform software development, and low level hardware development but has a primary focus on high level language development and interpretation. The project was started as a proof of concept that a high level language can be fast enough, flexible enough, and powerful enough for real world use. Over the course of the project however, a new focus was given on the importance of a unified development environment and how absorbing all parts of the development cycle into a single system can greatly reduce inefficiencies in the development cycle.

Introduction:

Developers suffer significant loss of time due to inefficiencies. Most of these inefficiencies fall into one of two categories, management and problems with what and how tools are used; Direct Abstraction aims to remedy the latter.

The total number of hours lost because of issues with either code bugs and toolchain issues is a serious problem. Large amounts of time are wasted reinventing the wheel, solving ever emerging bugs, and wasting time waiting for compilers and other tools to deliver the products required. Companies have gone to great strides to find ways to break down the problem with the toolchain by unifying all means of development in the work place. Two notable companies that have managed this are Apple with the Xcode development environment [6] and Epic Games with the unreal engine [7]. In both of these the user rarely has to leave the environment over the course of the entire project. Companies that have successfully managed to do this are able to develop a much more streamlined development process and gain a notable edge over their competitors. Microsoft also made attempts to unify all stages of development by adding the XNA library to the Visual C# development environment but has now given up on such attempts [8]. Besides a few rare cases a unified development environment is difficult to achieve and remains elusive to even the largest companies. Many languages such as Ruby and Python try to embrace paradigms that help reduce bugs. Adoption of these languages is slow and tools and libraries built around theme are sparse. This makes them less appealing to developers who rely on the large number of tools with C and C++ support.

Direct Abstraction is an attempt to try and solve these problems by offering cross-platform code and tools and a unified built in feature set. By creating a language based on other high level languages bugs such as memory leaks can be removed. [9]

High Level Overview:

Direct Abstraction is a high level scripting language that brings a shared code base to both the PC environment and ARM hardware (specifically the STM3240G-EVAL board as of current). Direct Abstraction works by offering an entry point for the user to execute code in the form of plain text. In the case of the PC version there is an interpreter that has been precompiled and is machine dependent that parses the user’s code in a text file and executes it. For the ARM device an already embedded virtual machine will look for code on an SD card on boot up to parse and run. In both instances this improves development time by removing the need for long compile times and having to deal with complex tool chains in order to get the code to build. This also gives the benefit of having a shared code base across multiple platforms. Code that would work for the PC machine will run on the ARM device. Due to differences in hardware the results vary based on what functionality has hardware support. Since the code base is the same for both the PC and the ARM device this means that the developer can write code for the end device without ever running the code on the device. Code can be rapidly developed on a PC environment with instant turnaround time for testing and then the final code can simply be ported to the final device and work.

Premises of Added Abstraction:

The user might think that the PC interpreter and the ARM virtual are on is the same but this is not true. The end developer and user might not realize that what's under the hood, nor should they care, but they are completely different machines that simply emulate similar behavior. The PC interpreter and the ARM virtual machine are completely different even down to the language they are written in (The PC's is built on C++ while ARM is strictly hardware supported C) but the underlying idea of how they work are the same. Direct Abstraction simply works by adding a layer of abstraction between the user and the hardware or system they are working on. This added layer of abstraction gives vastly more control to the interpreter as the user has no direct access to the hardware. The user never has to worry about whether or not "print" is supported on the target hardware or how to handle memory mapping, the interpreter/virtual machine will worry about that for you. This also aids in speeding up development by where you simply have to write the code you want and are sure that all the fine grit details are taken care of for you.

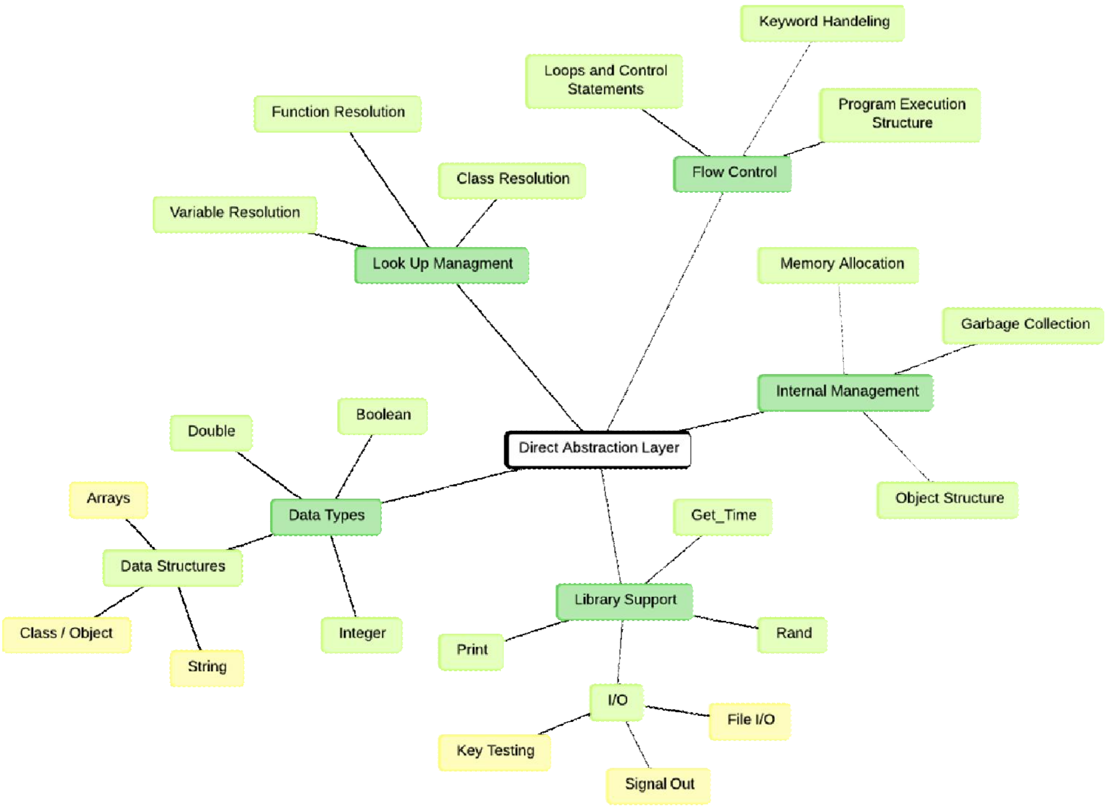


Fig 1) A mindmap of the how the abstraction layer works. By offer the following features the developer is removed from the inner workings and nuances of the hardware used by the end

De

Direct Abstraction is meant to replace the process portion of development and the execution stage. The act of unifying and simplifying the process is what allows Direct Abstraction to have such a large effect of development on the business side of things make it a valuable tool to help maintain cost and keep production efficient. Here we will discuss what is required to begin development with Direct Abstraction.

User Interface:

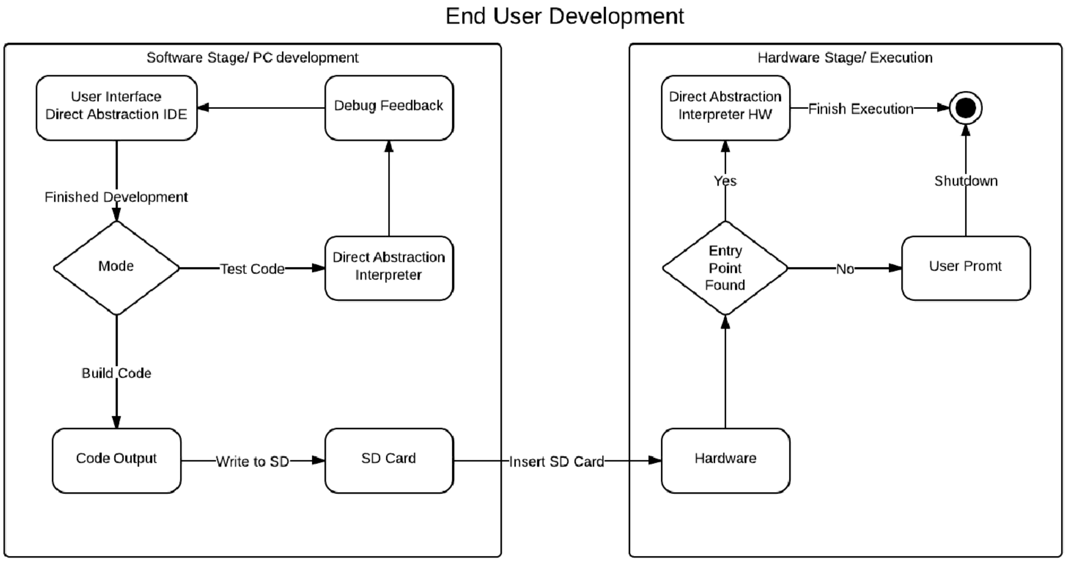


Fig 2) Development + User cycle for ARM hardware development.

End user d [illegible] stage, the Hardware/Execution stage, and an intermediary Delivery stage. Whether the user is developing for PC or hardware the stages are essentially the same where “SD Card” simply represents the distributables. During the development cycle a large amount of work is abstracted from the user allowing them to create the program without ever leaving the software/development stage.

The Programing Stage:

In the programing stage the developer writes the code. Here code is written into a text file by the name of “main.txt” which will be used as the entry point for the code. Any text editor will do as long as it can parse and save plain text files without adding any header information or distorting the file in anyway. The developer will write in Direct Abstraction Source (DAS). More can be learned about in the user manual section. When all of the code has been written is all must be concatenated to the main text file. Unlike C/C++ no other files will be linked during the compilation phase so all code must be located in the “main.txt” file. To test the code you can simply run the interpreter from the same directory as the main file and the code will be immediately executed. For rapid development you can bring the interpreter into the same directory as your work space and whenever you wish to test your file you simply need to run the interpreter. Once the programming stage is complete you’re main file will be your distributable. Currently there is no way to encrypt or compress the main file meaning for now. This means programs are distributed in raw text format.

Delivery:

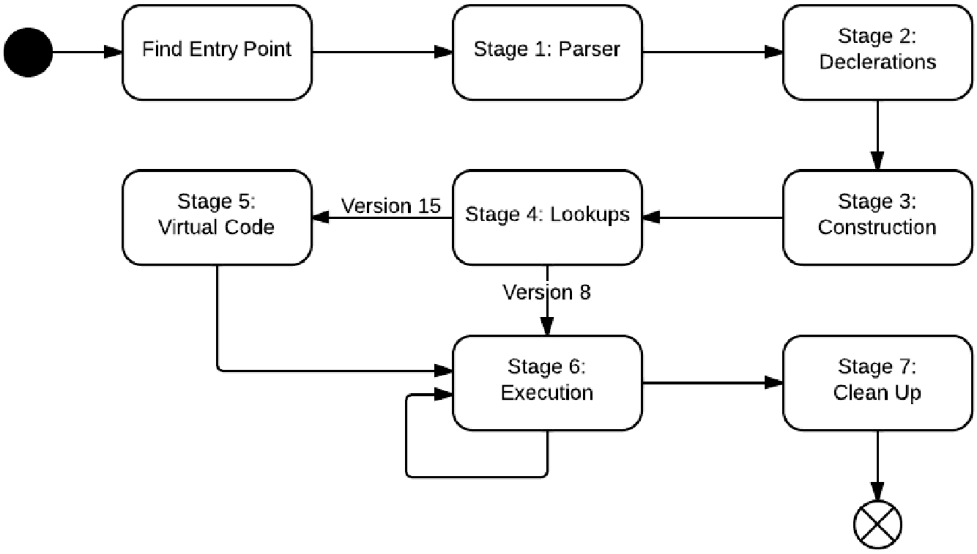
The delivery stage is made fairly simple. The main file is the only thing that needs to be distributed. There is no compilation to be done or any additional overhead. This makes it easy to patch as you simply need to deliver this one file to your users to update your Direct Abstraction program. On the PC side you must also deliver the latest Direct Abstraction interpreter along with your main file. Once the users have the version of the interpreter you are working with you no longer need to distribute. Updates and patches. This creates a lightweight solution to updating similar to dll files for C/C++.

Execution:

During execution code is parsed and turned into a syntax tree. In versions 8 and higher it is then compiled down to *bytecode*. Bytecode is a stream of characters that informs the interpreter which actions to perform. Each segment of bytecode is mapped at runtime to a set of assembly instructions to execute the code. In versions 15 and higher the bytecode is then compiled down into *virtual code*. Virtual code unlike bytecode embeds some assembly addresses right into the stream of characters. This lessens the number of steps the interpreter must take to execute the required machine code. By lessening the number of steps the interpreter has to take to execute machine code the less overhead it has. Code generation happens extremely quickly and is to only be performed once at startup. Bytecode and virtual code are never to be recompiled at runtime. Recompiling the code would add more overhead at runtime.

Error handling is handled here and any syntax errors or runtime errors will be presented at this stage. If the user happens to run into a runtime error a detailed block of information must be presented with things such as line of occurrence, type of error, time in execution of error, and possible fixes for the error. The interpreter will stop execution, clean up and then exit. If a syntax error occurs then it will continue to scan syntax but no code will be generated. This means multiple syntax errors can be produced but only a single runtime error will be. This feature can improve debugging of a large project as syntax errors tend to be numerous.

Interpreter



Design

Fig 3) All seven stages of a Direct Abstraction Interpreter. Each layer represents an overall phase.

The intrepreter has three phases: Parsing, Construction, and Execution. The first two steps format the code in such a way as so the interpeter can run the code. The third stage executes executed. In the parsing level the text is cleaned up by stripping out any useless information. This level is borken up into two sub stages; parser and declerations. The output is then used in the construction stage which constructs the logic flow of the program. Here execution and memory management are all preprocessed. This level is borken up into three sub stages; Construction, look ups and virtual code for versions 15 and higher. This preprocessing saves a large amount of redundent work from being done in the execution stage. The construction stage produces preprocsesed code called bytecode and then virtual code. Memory tables are preprocessed that contain constant information used at runtime. The common table entries include strings (“string pool”) and static arrays. The preprocessed memory tables are then passed to execution. In the execution stage the created bytecode is processed until the code runs to and end or an error occures. Once on of these conditions is met the program ends and cleans up memory and residual code.

Preprocessing – Stage 1:

On start up the interpreter or virtual machine will enter the parsing stage. This is the first of the seven stages to execute. The preprocessing stage cleans up the input code to make it easy for following stages to parse. To begin preprocessing the main file is first loaded into memory. Comments, extra white space, junk code, header files, and pound directives must all handled in this stage. The parser is unaware of what symbols are and merely preforms an algorithm on the text.

The algorithm is as follows. Read in a character and check if it is one of four types of characters. These types are number, text, arithmetic, and whitespace. All of these subsets are mutually exclusive. A comment section is considered any section between a sharp symbol (#) and a new line character. All text in a comment section is to be considered whitespace. If the character is white space replace it and all following whitespace with a single space character (value 32). If the section of whitespace contains one or more newline characters, replace the section instead with a single newline. If the character is not whitespace then it is preserved. If the character before the read character is not of the same type and either character is not whitespace then a single space character is added between them.

Two cases are handled specially. First, if a decimal character (.) is processed and both of its neighbors are numbers then the period processed as a number as well. Second, if the character is a quote character (") then all formatting is paused until a second (") character or a newline is found. This preserves the whitespace of strings. Quotes and decimal characters are treated as whitespace when they are in comment blocks. This process is continued character by character until the end of the file is reached. References to what type each ASCII character is can be found in the user manual.

Since no logic is ever processed here this stage will never produce a logical error. The only error handling the parser stage should preform is in the case that "main.txt" does not exist. As long as the main file is found this stage should execute perfectly fine without complaint. Errors such as mismatched quotation marks are to be handled in the third stage as syntax errors. Pound directives do not internally preform any logic but allow the developer to configure the interpreter in whatever way they desire within given parameters. More information on pound directives can be found in the user manual.

Declarations – Stage 2:

After the preprocessing stage a cleaned up version of the source code is passed into the declaration stage. Here a second pass is made to find out what user defined functions and classes exist in the code. First two global spaces are made for look ups which usually take the form of vectors. These two global spaces are for function definitions and class definitions. Each class definition can contain multiple function definitions.

The standard format for parsing is "def" or "class" followed by the name of the object. If the object is prefaced with "def" then it is a function declaration and the rest of the following objects are arguments to the function. A newline character is treated as the line delimiter and parsing should end after it is encountered. If no other objects are found after the function declaration then the function is assumed to take no arguments. Classes are not allowed to be followed by anything other than whitespace. If either declaration is prefaced with anything other than whitespace then a "BAD_DEFINITION" is thrown.

Symbol	Handle	Data (Fixed Bytecode)	Stage
BAD_ENTERY	Crash	"main.txt" is not found in directory	Stage 1
BAD_DEFINITION	Crash	Bad function or class definition	Stage 2
BAD_SYNTAX	Continue – End Stage	Double operator was encountered	Stage 3
TOO_MANY_ARGS	Continue – End Stage	Argument pushed into full function	Stage 3
TOO_FEW_ARGS	Continue – End Stage	Function evaluated with too few args	Stage 3
CONTROL_HANGING	Crash	End missing	Stage 3
FAILED_DEFINITION	Crash	Attribute or class in statement	Stage 3
AMBIG_LOOKUP	Recoverable	More than one possible lookup	Stage 4

Table 1) List of all Direct Abstraction Error Codes

Once a definition is found the code will be continued to be parsed until a terminating "end" is encountered. This means the parser must keep track of scope even though scope is not processed during this stage. This is to prevent an end used to terminate a control statement being perceived as terminating the function. Examples can be found in the user manual.

Subfunctions:

If the parser is in a class definition then subfunctions must be processed as well. If a function is processed as part of a class then it is added to the class definition space and not to the function definition space. If a function is declared in a function or a class is declared in a class then a "BAD_DEFINITION" is thrown. If two or more functions are defined in the same class with the same name then the newest declaration supersedes the older one. There is no standard implementation for closures as of yet.

Attributes:

Anywhere in the class the user can define attributes by prefixing a symbol with the "@" character. Attributes must be logged and added to the class that they appear in. Attributes are not function dependents and can be used in multiple functions owned by the same class. Only one instance of each uniquely named attribute is to be added to the class.

Appending Rules:

Once a class has been terminated then it can be reopened with another class definition. If the class definition already exists in the class space then it is reopened and new attributes and functions can be appended to the class. This means the order in which class definitions appear in the code effect which functions have precedence. This means the order in which files are appended to the "main.txt" is extremely important. In the specially built Direct Abstraction IDE (Integrated Development Environment) files are treated as a database and go in order from top to bottom. Another choice is to append files based on alphabetical order. Another good choice is to append them based on the age of the file. Because there are so many different ways a logical append can be done that makes the order of files essentially undefined Direct Abstraction only accepts one file which is "main.txt". This makes it so there is no ambiguity in the order of which declarations should go. Direct Abstraction editors however are free to merge any number of files into a single output of "main.txt". This means you should consult the specifications of the editor for appending order information. You can also avoid the issue altogether by programming to the standard and only programming in one file.

If any errors were thrown in this stage then construction will have undefined behavior. In the event of this the interpreter should immediately go to the cleanup stage (stage 7).

Construction – Stage 3:

In the construction stage all of the logic is processed. The construction stage takes the preprocessed codes as its input. It will also use the declarations previously created to decide what a symbol is. Construction involves three main phases called "symbol interpretation", "syntax tree construction" and "bytecode construction". Each phase will produce its own unique errors if any occur. This gives a more precise error message for tracking down the exact cause of the error. For each line of code syntax tree construction will run followed by bytecode construction. Each triplet of calls will produce a single line of bytecode. A line of code is defined as any range of characters between two newline characters.

Symbol Interpretation:

First we need to establish a symbol class. Since Direct Abstraction is weak typed any symbol must become any type at a moment's notice. Because of this a symbol must have information for every possible runtime time. This includes a pointer to other symbols, an enumeration for the type ID, a __int64 for the number data, a string class of some kind, a double for double precision floating point, and a boolean. A union can be used for most of the data types in order to keep symbol memory down. It is impossible for instance to use both the __int64 and the double so they can be put in a union to save 8 bytes. Symbol Interpretation starts by loading a group and advancing the code stream by the size of the group. A group is defined as any set of characters between two whitespace characters. Special exceptions of this are text symbols which are defined as any set of characters encapsulated between two quotes. This group is assigned to the string of the symbol. This data will be used for the rest of the processing steps.

The symbol is now given a type based on the characters it contains. The possible types used in Direct Abstraction include; string, double, number, boolean, function, class, keyword, container (Vector), variable, attribute, global and arithmetic operator. The simplest cases are checked first. If the symbol starts with a quote character then it is made string. If it starts with a dollar mark ($) then it's a global. If it starts with an "at" symbol (@) then it's an attribute. If it starts with a number then it is a double or a number depending on if this symbol contains a decimal character. If the first character is an arithmetic operator then it is classified as such. A list of arithmetic operators can be found in the user manual.

If the symbol does not fall into any of these categories it needs to be specially processed to see if it matches any of the keywords, built in functions, user define functions, or classes. To check if it's a user defined function or class the symbol is a user defined function or class it needs to be checked against all the function and class names registered. If it matches any of these types then it is to be typed accordantly. If it doesn't fall into any category then it defaults to a local variable and should be processed as such.

Now that a symbol has been typed and it has been given whatever data it contained it now needs to be give a priority value. A list of standard priority values is listed below. An updated table can always be found in the user manual as well. On top of those standard priority values the open parentheses ("(") add one full level to the priority value (usually 12) and the close parentheses (")") decrement the priority by one full level. It should also be noted the commas are a special character and the type they are given should default to a number and should have the same priority of a constant. The reason why will become apparent in the syntax tree construction phase.

Symbol	Priority	Symbol	Priority
Object	11	**Addition / Subtraction**	5
Range Operator	10	**Modulo**	4
Array Bracket	9	**Dot Operator**	4
Function	9	**Compare**	3
Couple Operator	8	**Greater/Less Than**	3
Power	7	**Assignment**	2
Mul / Division	6	**Keyword**	1

Table 2) List of all Direct Abstraction operation priorities

All of the current construction is simply so we can form the syntax tree in the next phase. At this point we need to actually extract the data from the code and store it into the symbol. For keywords, functions, classes, and variables this is simply the index of that symbol based on its memory mapping. When it comes to numbers and doubles the symbol needs to be parsed in character by character and converted into a base 10 numerical value and then stored in the symbol. For Booleans if the value is based on the text string "true" or "false" (case sensitive). The final type strings simply take the symbol as is with the quotation marks removed. A simple and efficient way to do this is with a memcpy or similar instruction call based on the binding language. The symbol is now fully constructed and can be inserted into the syntax tree.

Variable Registry:

Variables need to know how they will be treated in memory at execution time. Every time a variable or global is found it is checked to see if it's in its respective space. If it is then the variable will have its integer buffer set to its index in the memory space. If not then its name will be appended to the end of the memory space and its new index will be added. This means that the memory space for all variables is created when the page is in scope. Since Direct Abstraction does not support control statement scooping for variables all variables are either function, class, or global scoped. The memory space for function scoping is constructed at the start of each function and cleaned up at the end of each function. This is to make memory lookup and access for variables much quicker than standard hashing used by higher level languages. More about this design decision will be discussed in the execution section.

Syntax Tree Construction:

The purpose of the syntax tree is to make it so our code is translated into a format known as reverse polish notation (also known as postfix notation). This makes it so the code is free of execution priority and order. With this notation the Interpreter can create extremely quick stack-based bytecode. It is essential that the syntax tree be updated with each new symbol added to assure that the symbols are in the proper order.

The syntax tree itself is a lot like a binary tree. It has a root which is the start of the tree and any number of leafs. Each leaf will have a pointer to a left and right leaf so that the tree can be constructed. It should be noted that this is mostly because most operators have zero stems (a constant) or have two stems (most arithmetic operators). This changes when functions get involved where it could have more than two arguments. In this case it is easier to have an array of children rather than just two however the base of the algorithm works based on this left-right structure of leaf. This means it is simplest to treat index zero as left, index one as right and the rest of the indexes as function only arguments. It is also useful to have a parent node pointer, expectually when doing strength reduction however it is not required. In the syntax tree nodes closest to the root have the lowest priority with the root being the very last thing to be executed for this line of code. The further the node is away from the root the higher its priority. All leaf nodes must be constants. It is possible to write in rules to prune the leaf nodes by precomputing certain operations.[1]

The algorithm for adding nodes to the syntax tree is similar to updating a red-black tree. If there is no node for the root then the new node becomes the root. If there is a root then start the main check loop. Here if the node's priority is greater than the current node and the current node doesn't have a left child then this node is the node's new right child. If the priority is greater than the current node's and it has a right child then walk to the right child. Otherwise if the node's priority is less than or equal to the priority of the current node then this node becomes the new parent of the current node and the current nodes old parent becomes this node's parent. When the current node is made the child node of the new node it is made the left child. This means nodes only acquire left children by this less priority mechanic, otherwise nodes only move to the left. This is an extremely simple and efficient way to achieve O(n log n) when constructing the syntax tree and preserving the order of operations.

How this works for functions is instead of walking to the right child (index 1) you walk first to the left child (index 0). Every time you get a new node then you still walk to its left index. This changes when a comma operator with less priority then the function passes the function. When this happens the comma is not added to the syntax tree. Instead it increments the walking index for the function. In example let's say a function that takes 4 arguments is in the syntax tree and two commas act on the node. That means when the next symbol comes by it will walk to index 2. If a comma has higher priority than the function it is passed down the tree. This is to allow for stacked function calls to operate properly.

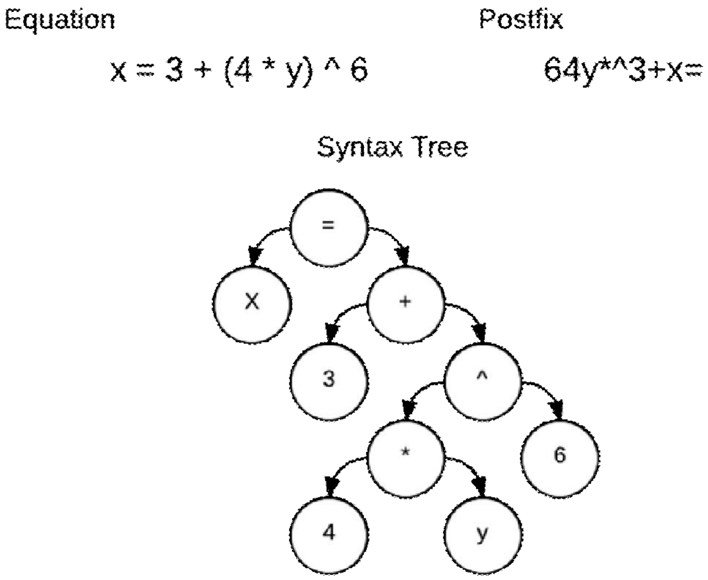


Fig 3) A sample syntax tree for the equation x = 3 + (4 * y) ^ 6. Also includes the postfix notation.

If two leaf nodes interact with each other than a "BAD_SYNTAX" error is thrown. This happens when two constants are back to back in code. An example of this is "3 + 4 5". Here the 4 and 5 will resolve to the same leaf spot because of the algorithm but since 4 is a leaf node and isn't allowed to have children and error will be thrown. If the symbol is a comma and the function it interacts with already has its max arguments than a "TOO_MANY_ARGS" error is thrown.

Bytecode Construction:

At the end of the line (new line character) the syntax tree is considered complete. The syntax tree is now evaluated to construct bytecode. Bytecode is abstract executable code used by languages such as Java.[5] When evaluating the syntax tree starts at the root. When a node is evaluated it first evaluates its children and then itself. This means the interpreter goes from lowest index to largest or in the case of most symbols, left then right. This means that the very last symbol to be evaluated is the root. This makes sense as based on our rules we are guaranteed that the root is the node with the lowest priority.

When evaluated the symbol needs to write a number of bytes into the bytecode stream. The bytecode stream is usually stored as an unsigned char* in order to make encoding simple. Each encoded bytecode instruction starts out with a two byte header which is also known as the flag byte. This byte informs the interpreter at the execution stage what operation it needs to perform. This also informs the interpreter how many bytes the code will have to walk in order to get to the next instruction in the set. The next few bytes contain the actual data that is to be processed. An example of this is a symbol that is a number and contains the value of 65. Since it is a number the flag bytes will be something like “n” to inform the interpreter that it is dealing with a number. Then we will have to encode the data or “A” (the ASCII value for 65). This gives us the resulting bytecode of “nA”. This is incorrect however since a number in Direct Abstraction needs to be 64 bits (8 bytes). This means we have to encode the whole number including the zeros which gives us “n\0\0\0\0\0\0\0A” (little endian). Sometimes the interpreter uses the endianness of the target machine. An efficient way to do this is to reinterpret cast the unsigned char* for the bytecode into the data type that you want to write into the stream. In this case we will reinterpret cast the bytecode stream into an __int64* and then write to the memory directly. This is a safer and easier option.

Symbol	Code	Flag Byte	Data (Variable Code)	Data (Fixed Bytecode)
Number	‘N’	0x4E	0x4E-xxxxxxxx-xxxxxxxx	0x4E-xxxxxxxx-xxxxxxxx
Double	‘D’	0x44	0x44-xxxxxxxx-xxxxxxxx	0x44-xxxxxxxx-xxxxxxxx
String	‘S’	0x53	0x53 “string”	0x53-0000000-xxxxxxxx
Variable	‘L’	0x4C	0x4C-xxxx	0x4C-0000000-0000xxxx
Global	‘G’	0x47	0x47-xxxx	0x47-0000000-0000xxxx
Attribute	‘A’	0x41	0x41-xxxx	0x41-0000000-0000xxxx
Keyword	‘K’	Ox4B	0x4B-xx	0x4B-xx00000-00000000
Built In Func	‘B’	0x42	0x42-xxyyyy	0x42-xx00000-0000yyyy
User Func	‘U’	0x55	0x55-xxyyyy	0x55-xx00000-0000yyyy
Operator	‘O’	0x4F	0x4F-xx	0x4F-xx00000-00000000

Table 3) A table of all the symbol and bytecode mappings for standard Direct Abstraction

Variables are special and the name is not stored but the index of the variable and its flag byte. For instance if we had a variable named “temporary” and it was the 5th variable to appear in the function so far then we would encode it as 0x6C (“l” for local) and 0x05 giving us a bytecode block of 0x6C05. This means variable are not looked up by name but rather by appearance order index. Globals and attributes work the same way but with different flag byte to tell the interpreter which memory locations it needs to access for the variable. For instance if we had the variable “$x” and it’s the 5th global then we would encode it as 0x67 (“g” for global) and 0x05 giving us a bytecode block of 0x6705. Notice that even though temporary and $x have the same index they are not the same variable. This is because the variables are located in two different memory locations. The number of bytes used to store variable indexes is up to the interpreter implementation but the standard is to use 2 bytes for variables allowing for up to 65536 variables to be used per memory space.

You may have noticed that different types of bytecode blocks have different sizes. This method of construction bytecode is called variable sized bytecode. The reason for this is you are only transcribing the amount of information needed to resolve the symbol’s data. Say you want to write a boolean that is set to true into the stream. You would transcribe into the stream 0x62 (b) and 0x01 (true) giving a bytecode block of 0x6201. Going back to the number with the value 65 we had the bytecode block of 0x6E000000000000000A. This is more efficient for space since we include the minimum amount of information to resolve the symbol however this makes it clunky at run time. At runtime the number of bytes we need to jump is now variable which makes walking the code and using control blocks much more complicated. A cure for this is to bad short symbols to get a constant size. This method is called fixed sized bytecode. Here our true boolean would code as 0x62010000000000000. It takes a lot more memory to store all of that padding but now all of our instructions are DWORD aligned which is much nicer and faster for the execution stage to handle. This means implementations have to make a memory speed trade off when encoding bytecode. Variable bytecode is currently the standard but this changes when virtual code gets involved (see the virtual code stage).

Once every node has been evaluated we will now have our code transformed into reverse polish notation and encoded into bytecode for execution. Now that it's done the interpreter encode a bytecode delimiter in place of the newline delimiter (the current standard is a single "]" in variable size bytecode). This delimiter will be used later to inform the interpreter when the line of code is finish executing a line so it can clear the stack. This makes it so that each line is its own instance of stack execution with a fresh stack being used for each line. Data may not be truly cleared off the stack however.

If a function or operator is evaluated and it has too few children than a "TOO_FEW_ARGS" error is thrown. A "TOO_FEW_ARGS" is also thrown if a function or operator is a leaf node. "DEVIDED_BY_ZERO" and a few other constant errors can be caught early here rather than at runtime. If any errors were thrown in this stage then construction will have undefined behavior. In the event of this the interpreter should immediately go to the cleanup stage (stage 7).

String Pooling:

It is possible to encode the entire string object into the bytecode but this is only doing able when using variable bytecode. In order to keep the bytecode a fixed length a string pool must be used. This applies for virtual code as well (see stage 6). This means a table of strings must be stored in the global space accessible at execution time. What is actually encoded into the bytecode is the index of the strings position in the string pool so it can be looked up at runtime and copied. This saves space for bytecode and can also be used for variable bytecode to keep size down.

Control Statements:

All control statements besides act like branches at a bytecode level. The only exceptions are "for" loops that do a little extra as well as branching. To know where to branch you must have your bytecode constructed at least to the branch point. Since any number of instructions can be in an "if" statement for example it's impossible to preempt where to jump. Because of this all control statements need to be stored in a table of look ups to be filled in later. The same thing applies to user function calls since we don't know where to jump into the bytecode yet. Since bytecode still has to be generated the interpreter simply generates a blank slot (0x00000000) and fills it in later with the proper address after the matching end has been processed. When an end is processed its address is used to fill in the slot left by the control statement. When a control statement evaluates a statement as false it will jump to this address. If the end matches a loop (while or for) then the end will have a data block of the address to the matching loop and will be encoded. If the end matches an "if" or an "else" then the end is not encoded (the "if" or "else" will jump here but do not need a jump back command). This encodes all of the branch conditions for the code.

Name	Type	Name	Type
If	Flow Control	**While**	Flow Control
For	Flow Control	**End**	Flow Control
Else	Flow Control	**New**	Symbol
Return	Symbol	**Nil**	Symbol
True	Symbol	**False**	Symbol
Def	Decelerator	**Class**	Decelerator

Table 4) a table of all the keywords for standard Direct

User Functions:

User functions cannot be resolved during bytecode generation. Since bytecode also act like calls on the assembly level you need to know the offset of the function in bytecode (bytecode address). Because functions might not have a logical order to them it is impossible to be sure if a user defined function is used after it has already been processed. Unlike control statements which only jump forward and ends that only jump back, a user function can jump to any place in the code. This means the best way to process user functions is to write the header flag byte and an empty slot “0x00000000” followed by two more bytes “0x0000” (four bytes for fixed bytecode). The second set of bytes is used to resolve the actual number of arguments the function needs (this is unknown until the function is processed). The name and index of the function called are stored in a table along with the address in bytecode. The table will be used to fill in the blank during stage 4. This will be the address to call and run when the function is called. When a function prototype is processed (“def” keyword) the function name and address are stored in a second table. This table will be used to resolve the first.

Look up tables – Stage 4:

Now that the bytecode has been fully created all of the user defined functions need to be processed. To do this the two tables created during bytecode generation are used. Every element in the function resolve table will be looked up in the second table and its blank slot will be filled with the address stored in the second table. The last two bytes need to also be filled with the number of arguments that are to be passed to the function. Every single function call needs to be resolved. In order for the bytecode to be considered complete. Once all of the look ups have been resolved the bytecode from this point on is considered executable.

Virtual Code – Stage 5:

At execution time each set of flag byte needs to be resolved, usually by a large case statement. Once the statement is resolved it may need further processing before the operation can be executed. Common cases of this double processing are arithmetic operators. When an arithmetic bytecode is indexed the interpreter must first resolve which bytecode operation it is. After that it needs to also resolve which arithmetic operation it is. Once that is all done the operation is finally executed and the bytecode is indexed by a variable amount (or nine in the case of fixed bytecode). Virtual code optimizes all of this by jumping straight to the function required to execute the code. This is done by encoding the proper function pointer

Virtual code is an optional stage and isn't required to perform execution. To begin the bytecode is reprocessed into a fixed format of twelve bytes. This makes it so every bytecode operation is twelve byte aligned. It should be noted that even standard fixed bytecode will need to be expanded by two bytes since fixed bytecode is only nine byte aligned. The data blocks are expanded into fixed format (padded with zeroes) and the flag byte are expanded too four bytes. Once the expansion and formatting is complete the virtual code can be processed. Here the four header bytes are simply replaced with a function pointer. This can be done by doing a few special "reinterpret_cast" when writing and calling. The mechanic can be found in the source code for the C++ interpreter however the ability to encode functions into the bytecode stream varies greatly based on the language and system the interpreter is written on.

To make this possible a few things are need. First of a family of functions will need to be created all with the same signature. One function is required for each one of the bytecode operations as well as one for each of the arithmetic operations. The standard signature is "void func(virtual_data*)". Virtual data is a structure that contains all of the information required for any operation to execute. The structure contains a pointer to the current stack, a pointer to halt execution (usually a boolean), a pointer to the local variable buffer, a pointer to the current index, and a pointer to the virtual code (usually a char* though other data types can be used to write virtual code). This structure is constructed at the start of every execution call and will be used passed to every operation call make it extremely fast.

When encoding the virtual code every set of flag byte are mapped to their corresponding function which is encoded as the new "flag byte". An example of this is our number 65. The bytecode for this symbol is "n0000000A". The function that pushes a number onto the stack is called "Number_VCode" so its address will be what is encoded into the virtual code. This gives a virtual block of "xxxx0000000A"; where "xxxx" is the address of Number_VCode. It should be noted at this point that function addresses change based on how the interpreter executable is loaded into memory by the OS. This means that the address value ("xxxx") will be different every time that the interpreter is run. This runtime variance means that virtual code unlike bytecode cannot be preprocessed and save for later execution. Virtual code is not only machine dependent but also instance dependent making it useless after execution.

It may also be noted that each step bring our code closer to looking like assembly. This is no coincidence as the simplest way to get execution on computer architecture is an assembly like format. This means as closer we get to raw machine instructions the less overhead we get. The problem with this is our virtual code becomes less and less readable to humans making it harder to debug. The code we execute also becomes more and more complicated and harder to manage. No further processing of code is done because of this and Direct Abstraction stops at virtual code. While it is possible to stop at bytecode or compile down to machine code, virtual code is a good tradeoff between speed and maintainability. Virtual code helps bridge the gap between interpreter and compiler.

Execution – Stage 6:

Execution starts by allocating memory for all of the variables that will be used during this execution block. The stack is also created with some initial slots. Execution then varies depending on wither virtual code is being executed or bytecode. If virtual code is being executed then all the execution block actually runs is a "while" loop with a single call to the first four bytes of the current bytecode instruction (bytecode + index).

If bytecode is being executed however then inside the while loop a case statement needs to be created. In the case statement each flag byte is processed and its functionality preformed. In the case of the operator flag byte the code is off loaded into a second function that has a second case statement that decides which operation will be performed.

When an operation is preformed it may take things of the stack. The symbols are taken off the top of the stack. When done operations must always push at least one symbol back onto the stack with the exception of the clear operation which clears the stack. Operations such as numbers, variables, and strings, do not take anything off of the stack. If a function or operator did not produce a symbol then a "nil" symbol is pushed on the stack instead. After the operation the index is incremented to advance to the next bytecode; twelve for virtual code, nine for fixed bytecode, and a variable amount depending on the instruction for variable sized bytecode.

When bytecode has finished execution the final symbol on the stack is returned and all the memory is cleaned up. Here garbage collection kicks in and all local variables need to be torn down and all of their memory freed. This makes it so garbage collection is free and localized. Garbage collection is similar to constructors and destructors in C++. It should be noted that the interpreter needs to keep track of the number of objects pointing to pointer types. If nothing is pointing to the object it is safe to free it.

Calling User Functions:

Users functions aren't as simple as a branch like control statements. Instead when a user function is called a new instance of the execution stage is called with the new index being the index of the user function in bytecode. The new execution stage runs the function call and the arguments of the function all are passed as local variables. The new execution stage runs as stated above and can even spawn other executions and can even call a new execution on itself (recursion). This makes it so all local variables and such are self-contained to the function. When a new execution is spawned it takes care of its own local space build up (including the locals passed to it as arguments) and maintains its own stack. On exit the execution is responsible for its own clean up as well and should return the last symbol on its stack. If the base function ("def main") returns anything then it should simply be discarded.

Cleanup – Stage 7:

Cleanup is the final stage of the interpreter. Here all of the global memory usage is cleaned up and everything is checked to make sure it's consistent. Since pointers are not a problem and arrays prevent overrun and underflow, there isn't much to report at this stage. Execution times and other logs can be transcribed at this stage if desired but it isn't necessary. In interpreters written in C++ or other languages where destructors or garbage collection exist it is very possible the cleanup stage will contain little if any code. It is possible (and useful) however to use information the interpreter gathered at runtime to generate even better code and write it to a file for the user. Cleanup has far more information about runtime behavior than any other stage besides execution. Unlike execution however, cleanup is allowed to be slower since it has no runtime penalties the user will overly notice. A good use for this is cleaning up what are known as "hot loops" and other commonly executed code bits with faster code alternatives. This saves time during future executions of the code.

This is all extra however. In practice cleanup's only responsibility is making sure all allocated memory is gracefully taken care of and the final bit of book keeping and error checking comes back clean.

Optimizations:

There are many optimizations that can be performed to speed up Direct Abstraction. A lot of optimizations are baked into the design decisions of the interpreter itself and are there for considered standard optimizations. Standard optimizations are one of the major differences between Direct Abstraction implementation versions. There is however still a large amount of room for nonstandard optimization that Direct Abstraction makes available to the developer. Over time optimizations tend to move from nonstandard to standard as more support is built for them. An example case of this is Virtual code execution which is not fully standardized yet but is moving in that direction with version 15. It should be noted that all optimizations should not change end user Direct Abstraction behavior. No matter which set of optimizations are used the interpreter must still be compliant with the behavior rules stated in the user manual.

Differences between Interpreters:

The hardware interpreter runs as its own virtual machine. The virtual machine is very similar to running the Java Virtual machine.[5] The hardware interpreter had its own set of challenges. All of the functions had to be mapped to library functions on the hardware. This was achieved by using peripherals on the hardware. USART over RS232 was used for input. The LCD screen attached to the board was used for "print" and "print_color" functions. Still there were a large number of library functions that have of yet to be mapped. These differences prevent the software interpreter from mapping one to one with the hardware virtual machine. Limitations and challenges with peripherals are preventing features from being ported to the hardware for now.

DA Compiler:

Direct Abstraction also supports compiling down into native machine code. This is possible once the virtual code has been generated and all control paths resolved. Each virtual code is translated into a set of memory and operation assembly instructions. Once the assembly instructions are constructed the code is compiled into machine code and filled into a buffer. Currently runtime ahead of time compiling is the only means supported. This requires the interpreter to layout memory maps and does not support dynamic linking. This means all calls supported are direct far calls to addresses and not resolved jumped tables.

Two main problems prevent direct mapping however. The first is the complexity of Direct Abstraction operations. Even simple instructions such as the addition operator ('+') is resolved with as many as a hundred x86 assembly instructions depending on the version used. This means the complexity involved in transcribing Direct Abstraction code to machine code is a lot more complex than transcribing languages such as C and C++. This mostly has to do with the weak typing system however garbage collection and built in data structures add to the complexity as well. To resolve this complexity in a manageable fashion machine code is transcribed in standalone batches of assembly instructions per DA instruction. This however limits the ability to optimize the machine code.

The second problem arises from the fact that Direct Abstraction bytecode and virtual code operate on a stack architecture while machine code for chips such as the x86 architecture are most efficient with a register architecture. This means special considerations need to be taken to assure optimum use of registers. A common example is handling of return values from operations. Rather than push return values onto the stack to instead keep them in EAX and immediately use it with the next operation avoiding an intermediary. There are a number of other considerations including stack and function memory layouts. For the stack it is more efficient to simply move ESP by the number of bytes that will be used in the function rather than make room for them as you need more. This is possible since all variables are function scooped and thus the total local variable size is known at the start of the function. For functions it is easier to use the method described above is to let the interpreter load the functions and simply encode a far CALL to the function pointer address.

Benefits to Compiling:

Compiling the dynamic language is the final step in the execution pipeline. Like the other execution optimizations (such as bytecode and virtual code) this step provides a boost to raw execution. Compiling DA down to local machine code vastly reduces the amount of machine code instructions that need to be executed per opcode. Compiling DA doesn't change how efficient opcodes execute but reduces a lot of the overhead of maintaining execution. When working at the machine code level optimizations can be taken such as changing from a stack based architecture to a type of register architecture. Working at this level also gives you direct control over the stack and function calls. This gets the execution closer to the CPU and removes unnecessary overhead. At this level the only thing preventing the language from being as fast as C/C++ is the added overhead of the opcodes themselves and how efficient they are. The only optimizations that remain past this point focus solely on reducing opcode complexity and overhead.

Compileability of DA:

DA has a number of design features that make it easier to compile down to raw machine code. The first is the ability to know the size of the stack at the start of a function and know that the size will be constant. This is due to the fact that all variables in DA are function scooped. Symbols themselves are also a static size (usually 0x30) make it very easy to manage stack operations. The symbols also have an initial state of all zeroes making it easy to construct symbols by simply memsetting the stack frame to 0x0. Function calls also accept symbols as only pointers under the hood making it extremely easy to pass functions from one context to the next. Types are also made easy to manage by simply moving the relevant type byte into the right position. Symbol to symbol assignment is easy to manage as only nine bytes need to be copied including eight data bytes (regardless of type) and the type byte. It is also possible at this level to reduce symbols down to only 12 bytes (11 bytes on 4 byte alignment) with a tradeoff of slower execution speed.

Several features however are much harder to map to static code. Large amounts of metadata need to be pooled in order for proper execution (strings, doubles, floats, addresses, ect.). Also garbage collection needs to be executed at the right times and simple missteps in the compile stage can create memory leaks that compound with execution time. These problems however can be minimized by keeping symbol size at 0x30 bytes allowing data to be kept around until collection time rather than having to keep track of the data constantly.

Compiling Benefits:

The most noticeable benefits come from memory access instructions such as assignment which can be reduced down too as little as three move instructions. Other instructions benefit greatly from the increased locality of data and not having to move as much around constantly. The greatest overall benefits however come from the fact that you are no longer jumping in and out of functions. With bytecode or virtual code execution you must constantly call out to memory locations to preform execution and then return simply to go to another location that contains the next instruction. By encoding all of the execution code linearly you stop throttling the code by adding unnecessary look ups for the execution code speeding up every opcode by at least some factor.

Direct Abstraction Encryption:

Direct Abstraction supports a simple encryption algorithm based on a reduced SHA-2 variant. While nowhere near as secure as SHA-2 it is extremely quick. This is important as extra work to processing the file would cause the startup of the program to slow down adding undesirable wait time between each execution of the file. The method makes use of a private key used by the interpreter which is hardcoded for each interpreter implementation. This makes it so encrypted code can only work on a certain interpreter family which aids to prevent attacks to decrypt the file. The algorithm also uses a number of features such as random keys and feedback to help prevent attacks.

Testing Model – SOLID Principles:

Direct Abstraction is split into separated stages to help aid in testing and expanding the interpreter. All stages of the interpreter are setup in such a way to have minimum dependencies on each other. This makes it possible to test each individual stage with its own set of unit test. Each stage only requires certain structures such as the global variable map and string pool which can easily be mocked and tested using tools such as gmock. Mocks are objects that can be made to create dummy functionality for additional dependencies. The only dependence that can’t be mocked is little more than a character stream to be processed and most stages only output a character stream. This makes it easy to map a certain input to a proper output aiding in testing methods.

Virtual code if used also added an extra benefit to testing. The execution stage quickly becomes the most test heavy part of the interpreter as the complexity grows. Virtual code aids in execution testing by separating out all executable functions into standalone functions. This makes it possible to test every single execution model in separate to make sure it has the proper output. Gmock helps by allowing mocks to be created for the stack and locals.[4] By mocking the locals and stack it is extremely easy to test each operator with all possible combinations. Variables and memory access can also be managed relatively easily by creating mocks for all other memory spaces besides the one being tested to make sure that access is being managed accordingly.

The hardest systems to test regardless of setup are control statements. Control statements manage flow control and there for require some kind of code to move around and operate on. It is very hard to make sure construction of control statements and execution of control statements work without testing them together. This makes integration test the prime test for control statements. A side effect of this testing is it increases the number of dependencies and points of failures. This means control statements tend to be the most common point of failure during stability testing. “For” loops tend to have the largest amount of dependencies as they require a maintaining and construction of a memory space, conditional evaluation, and flow control. All of these dependencies make having full code coverage in testing nearly impossible. Large amounts of focus should be put on testing control statements when creating a Direct Abstraction compliant interpreter.

Alternative Testing:

Since Direct Abstraction is setup to follow the “SOLID” principles there are a number of testing frameworks that are available for use. Each function can be tested separately but it is still recommended to use some form of mocks and fakes if gmock is not available. If mocks or fakes are not usable in the language of choice used to back the language then extra care for testing should be used. Since the language has an infinite number of inputs and an infinite number of outputs testing the language to make sure it works is extremely important. Maximizing code coverage is a must with all of the pieces being separate. This is the limit the number of failures that can compound when parts of the interpreter framework work together. Without the ability to make sure that each operation and stage works by itself, interpreter construction and expansion will quickly become extremely hard to manage and debug.

For hardware it is recommended to have some kind of output such as RS232 to make sure behavior is correct. Tools such as Keil
was a simple stress test program. The stress test program attempted to have full test coverage. uVision can be used and offer minimum debugging environments.

Test Driven Design:

While gmock and unit testing covered things such as memory leaks and improper behavior it doesn't cover real world usage. A number of files were created during the course of development to test real world applications. Test programs such as Fibonacci number generators and simple guessing games were created to test control statement flow. Simple programs similar to "hello world" were also generated as new features were created. Simple sample programs are one of the best ways to do integration tests. While they are much harder to debug when an error occurs they make sure to test throughput of all stages.

The most useful program created whenever new features were added they were appended to the file and tested. This test helps to make sure the interpreter is stable under large workloads. The stress also has a loop that stresses loading and writing data, loops, conditionals, and math functions for over a million iterations. This test helps also drive Direct Abstraction optimizations aiming to make the language faster and more efficient. This creates a testing balance between features and speed. Some versions focus on making the language more robust and stable while others are targeted only on speeding up the language.

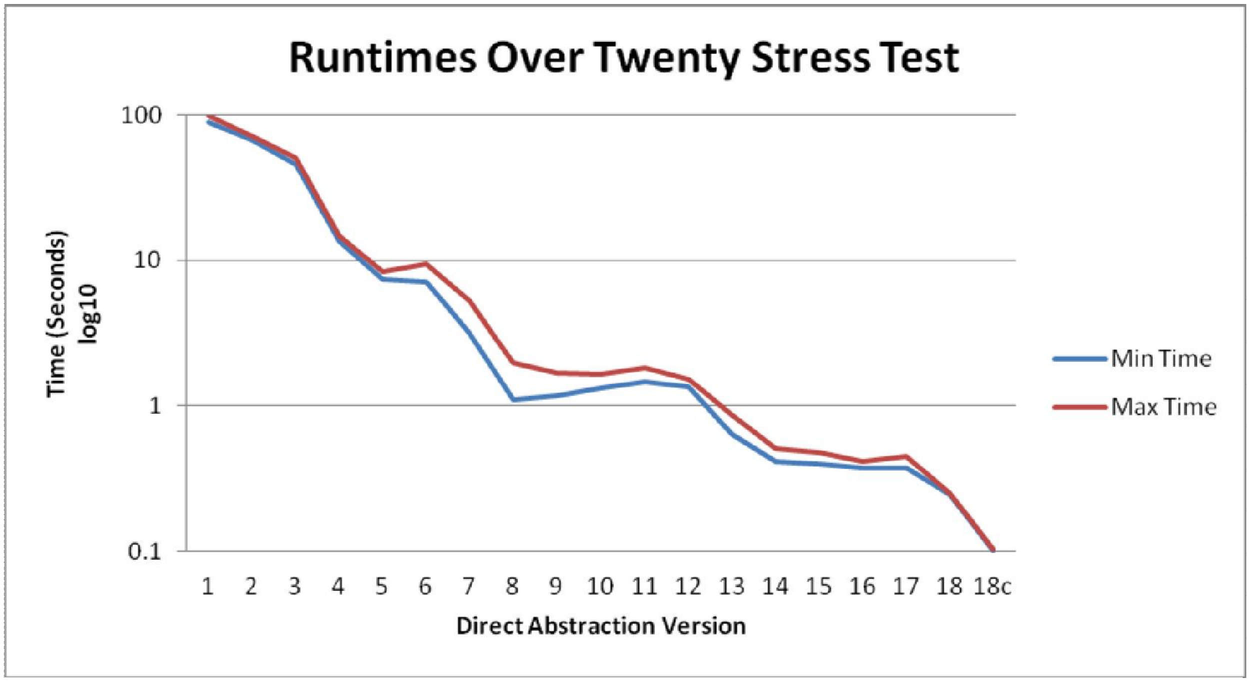


Fig 4) A graph of testing times for the stress test across all Direct Abstraction versions since DA 1.0. Execution was tested on an Intel Core i7 M620 @ 2.67GHz with 4 GB of ram on 64bit Windows 7 SP1.

A good majority of the speed improvements come from the use of better methods for executing the stack. The first notable dip is from version 3 to 4 which was the move from raw execution to tree execution. The second dip between 6 and 8 was the changeover to bytecode. The small dip from 12 to 14 was the introduction of virtual code. The final dip from 18 to 18c was the introduction of the AOT compiler. It's not always as simple as improving the method of execution for pure speed ups. A lot of speed improvements come from decreasing the foot print of operations to allow more opcodes per second to be executed. This is done mostly by finding tricks to reduce the amount of information that needs to be encoded or decoded for any given operation.

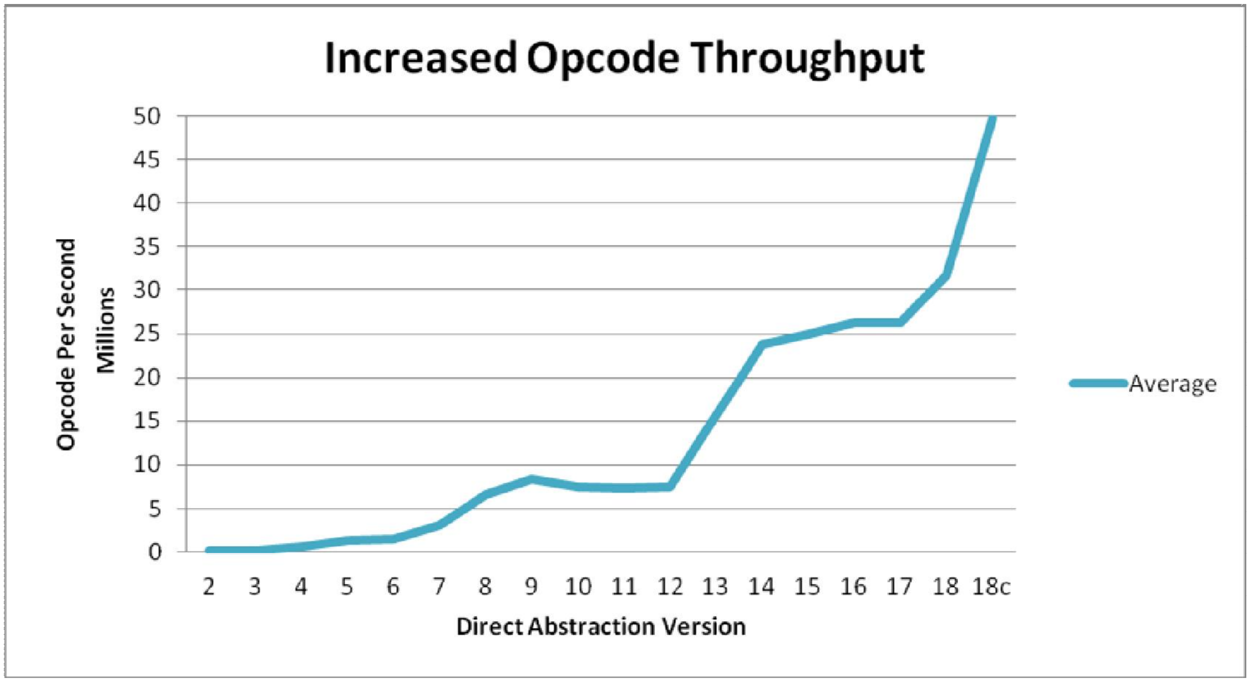


Fig 5) A graph showing the rapid increase of the number of opcodes can be executed per second. Execution was tested on an Intel Core i7 M620 @ 2.67GHz with 4 GB of ram on 64bit Windows 7 SP1.

Going Further with DA:

Direct Abstraction is an ever evolving standard that grows and changes to the challenges that need to be. The next step for Direct Abstraction is the addition of more interpreters for varied platforms to enable and improve cross-platform ability. Additional features to existing interpreters are also planned including graphical bindings for the Window's interpreter.

Besides additional interprets a google chrome plugin is in the works. An operating system with a shell script that is Direct Abstraction compliant is also planned to make Direct Abstraction scripts executable at an operating system level. Mobil development has also been experimented with and interested is expressed in creating an Android interpreter. Mobil development however still seems extremely unlikely at this time. If an operating system is not pursued for now then a second ARM virtual machine will be the next major development extending embedded development beyond the STM3240G board.

Conclusion:

Direct Abstraction has done a lot to bridge the world between standard PC development and hardware development. By providing a common interface for multiple environments it is vastly easier to develop cross-platform software. Standard interfaces remove the need for expensive tools to develop as well vastly reducing the barrier of entry.

By taking advantage of several design decisions Direct Abstraction remains flexible and fast while relatively easy to implement. With virtual code, different bytecode modes, and several optimizations to choose from the Direct Abstraction standard gives large amount of room to make an interpreter to meet the needs of an environment. Lower memory options for symbols and code size are available make Direct Abstraction useful for embedded hardware and web development. Implementation specific optimizations and virtual code make it possible to make large speed gains for use in games. A complex abstraction layer of built in functions also allow for large amount of code offloading onto native language bindings allowing complex code to be speed up and simplified. All of this backed with the development friendly higher level language features such as weak typing Direct Abstraction can be formed into a great fit for just about any platform and any environment.

Acknowledgements:

Thanks to Robert Kevin Secretan and Robert Gervais for aiding in direction of the language's syntax and supplying numerous resources to aid in the development of the interpreter.

References:

[1] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers, Principles, Techniques, and Tools. Reading, MA: Addison-Wesley Pub., 1986. Print.

[2] Abelson, Harold, Gerald Jay. Sussman, and Julie Sussman. Structure and Interpretation of Computer Programs. Cambridge, MA: MIT, 1996. Print.

[3] "How Kernel, Compiler, and C Library Work Together." - OSDev Wiki. N.p., 19 May 2013. Web. 25 Sept. 2013.

[4] "Googlemock - Google C Mocking Framework - Google Project Hosting." Googlemock - Google C Mocking Framework - Google Project Hosting. N.p., n.d. Web. 04 Dec. 2013.

[5] Meyer, Jon, and Troy Downing. Java Virtual Machine. Cambridge [Mass.: O'Reilly, 1997. Print.

[6] "Apple Developer." What's New in Xcode 5. N.p., n.d. Web. 07 Dec. 2013.

[7] "Unreal Game Engine Technology." Unreal Engine News RSS. N.p., n.d. Web. 07 Dec. 2013.

[8] "Microsoft Has 'no Plans for Future Versions' of XNA Software, Will Not Phase out DirectX | Polygon." Polygon. N.p., n.d. Web. 07 Dec. 2013. <http://www.polygon.com/2013/1/31/3939230/microsoft-has-no-plans-for-future-versions-of-xna-software>.

[9] "C Programming: Programming Languages, an Introduction." - Wikibooks, Open Books for an Open World. N.p., n.d. Web. 02 Oct. 2013. <http://en.wikibooks.org/wiki/C++_Programming/Programming_Languages>.