

1 Definitional Architecture Metaprogramming

"The idea that you can do a programming language in 100 000 lines of code or you might be able to do it in a page and there are advantages to the page for sure" –Alan Kay 2009 How Complex is "Personal Computing"?

Definitional-Interpreters (DI)– convey the ideas and semantics of a language with brevity and clarity; the advantage with defining a language with this method over a specification document is that the result of the **DI** is executable, so there is no question of the meaning of an expression in a context. The need for avoiding complexity is more important in the context of **DI** than the need for optimization.

Architecture– Different cpus need different executables if they use different instruction-sets, and even if they do use the same **ISA**(instruction-set-architecture), they may have new Extensions that may extend an **ISA**, to let older programs run on newer cpus.

Metaprogramming– a programming technique in which computer programs have the ability to treat other programs as their data.

"C" Compiler (`damcc`) as the DI of the "C" language – Rarely is a language as well put together as `C`, Mixing `C` and assembly can often be as simple as not translating the assembly parts of the `C` code. A lot of compromises were made to do this; For all intents and purposes `C` is not so much a language so much as it is basically assembly and just as dangerous. `C` is a notoriously "Dumb" language, in that it will do what exactly what **you** tell it to do; Including the set of all known computer *bugs*: memory leaks, stack overflows, disk destruction, or worse.

Architecture Data-Structure "C" – For compiling/interpreting any language; `C` is an excellent target architecture to use as a stepping stone to actual compiled assembly code. "`damcc`" is not just a **DI** for `C` it is also to serve as documentation for the different **ISA**s. Users should be able to understand an architecture quickly and easily by reading the code for the supported **ISA**s.(as well as elf files and static/dynamic linking) *"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."* –Alan Perlis

Stage 1 subset of C as an intermediate language– Define what the `C` language is by using "Ometa" to convert from preprocessed `C` to `C` or our small subset of it that is more easily translatable to machine code. The goal here of using this subset of `C` is to "explicitly enumerate all the nuances of the language" by "removing them".

- The meaning of all loops translated into gotos
- nested if statements translated into guard clause non-nested if statements
- no more Arrays, Types, Struct, Union, Enum, or any other data-types not manipulatable with assembly; just pointer arithmetic.
- even though we can replace function calls with gotos, they need to be left in to represent where we will leave symbol references for the "`.o`" files
- copies of variables that are created by a function call if they aren't passed by reference, the goal here is to perhaps be Bi-directionally translatable.
- `sizeof()` tells the number of bytes that type or array occupies, and `sizeof()` is known at compile time, and most of the time this number is fed into arithmetic functions that can also be calculated at compile time; which can allow optimizations here.
- `#include` lines willl be removed, files referenced by `#include` will have their funtions scanned and added to a table of functions so that when a external function is called the appropriate "`symbol()`" call of the elf file can be made.
- **Memory–** all of variables **AND** pointers Each function is to have all memory explicitly allocated/delocated with appropriate calls to:

  malloc() realloc() calloc() aligned_alloc() free()

memory continued– Malloc() basically wraps both brk and mmap2 syscalls, but it also keeps internal information about the memory that has been allocated for use in later malloc calls or free calls. how this is implemented falls outside the scope of a compiler: Having to move around countless copies of the malloc() c function in each and every executable in the operating system is going to be slower than Dynamically linking to malloc(). Executable files aren't just a bunch of assembly code, there needs to be dynamically linkable functions at execution time even if we do our own linking.

"I came to realize that if I'm not writing a program, I shouldn't use a programming language. People confuse programming with coding, coding is to programming what typing is to writing. It's something that involves mental effort, what you're thinking about, what you're going to say, the words have some importance; but in some sense that even they are secondary to the ideas. In the same way programs are built on ideas, they have to do something and what they're supposed to do, is like what writing is supposed to convey. If people are trying to learn programming by being taught to code, well they're being taught writing by being taught how to type, and that doesn't make much sense."
–Leslie Lamport https://youtu.be/rkZzg7Vowao

1.1 OMeta Metaprogramming

"The system that tries to find coverings of those trees pieces of structure within the trees is itself very very simple and you can write 2 functions that do it reduce a particular piece of tree to a quantity a register in the machine in memory location literal in the machine or avoid just a statement that returns no value and do that by matching a particular piece of tree against the pattern in the list of rules these things are recursive so that structuring the trees can have sub structure etc and what falls out at the end is a yes or no answer and it's the inverse of the inverse of traditional parsing if you like where a normal posit X unstructured free text and creates a structured representation

```
reduce(tree, startSymbol) =
    foreach rule in startSymbol.startSets
        if match(tree, rule.pattern)
            rule.action()
            return startSymbol
    return false

match(tree, pattern) =
    if (pattern.isSymbol()) return reduce(tree, pattern)
    if (tree.first != pattern.first) return false
    foreach treeElement, patternElement in tree.tail, pattern.tail
        unless match(treeElement, patternElement)
            return false
    return true
```

what this bottom-up system does is it takes a structured representation and tries to find inside quotes an optimal unstructured equivalent which in our case are machine instructions for the for the target machine

One of the interesting things that we've now done is we've described cogeneration as a tree rewriting process and what we've really been doing throughout the whole talk is describing how various phases of the implementation of a programming system can be described in terms of simple tree line structures and rules rewriting rules applied to them and we can hope in the end to create a single very small very easily understandable system tree rewriting system in which the whole of the language implementation problem is described.

now if you choose to go our way and play with some of the ideas that I'm talking about today one thing you will discover is that you need to find a fixed point to get the recursive descriptions started and the way that we did it was

write a bootstrap compiler in C that could understand a little object language we then rewrote the object language in the object language fed it through the bootstrap compiler to create a real compiler and then deleted the bootstrap compiler because it was now uninteresting.

Then everything else the the transformation rules for taking structures applying all the functional rules to them to create executable things within fed in and from that point on the whole system takes off exponentially because of the simplicity the expressiveness that is built into those 2 mutually supportive models within it."

– 45:36 into presentation "Building Your Own Dynamic Language" by Ian Piumarta slides also see Alessandro Warth - 34:20 - How Complex is "Personal Computing"?

Metaprogramming Example translator Control flow operators? maybe all we need is "goto" read this code:

```
while (condition) statement
//the above while-loop is equivalent to the following logic
begin:
if (condition) {
    statement;
    goto begin;
}

do [loop body statement] while ([condition]);
//the above do-while-loop is equivalent to the following logic
begin:
loop body statement;
if (condition)
    goto begin;

for (variable-declaration; condition; variable-update) statement
//the above for-loop is equivalent to the following logic
variable-declaration;
begin:
    if (!condition)
        goto end;
    statement;
    variable-update;
    goto begin;
end:
    // end loop

switch (expression) {
    case const-expr1: statement1
    case const-expr2: statement2
    case const-expr3: statement3
```

```
    default: statements
}
//the above switch is equivalent to the following if-else logic
type value = expression
if (value == const-expr1)
    statement1
else if (value == const-expr2)
    statement2
else if (value == const-expr3)
    statement3
else
    statement0

#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
#                                                                   #
//something else
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

1.2 (ELF Format) Relocatable Object "`.o`" Files –

Relocatable object files are files that are in the "elf" executable and linkable format. To Dynamically link to each foreign function call requires parsing header files to see which header provides each function that is used in the current file, to properly call the function and throw an error if a function isn't available. "`.o`" files are binary data assembly with symbols packed in, based on a specification. This is the format for linux executables and libraries, These files are the raw inputs to "linkers". Translating from "`C`" to "`.o`" linkable binary files is the goal here and the definition of compiling.

1.2.1 Stage1 notes

***"Clean"* and convoluted–** Recall from the introduction after Stage1 is complete there will be no need to:

- parse the `sizeof()` command
- parse Arrays, Types, Struct, Union, Enum
- parse control flow statements switches loops etc.
- create any variables – the c code will be requesting memory for it's own variables and pointers the appropriate function calls (`malloc()` etc.).

1.2.2 Stage2 - on the shoulders of giants

58:57 - 60:54 into Cliff Click - HotSpot "C2" JIT "The Java Hotspot Server Compiler" high level overview of how C2 works Click, Paleczny, Vick

*"**Outline how The architecture description files, in how in general made the compiler suitable for many platforms all at once:** So there are some things there that I wouldn't do again including the burrs technology there is an architecture file that is used to describe a CPU architecture including the set of the kind of instructions and a mapping from the sea to ideal nodes to machine equivalents and the registers that are allowed and the encoding for the machine code encodings for them is sort of the common set of things there's a bunch of the things that are in there that are all these fine-grained details but the big pieces are:*

here's a mapping from idealized nodes which is what the eye are mostly runs on to get under code gen to hardware instructions and the registers that are allowed both on each individual hardware instruction input and overall what are those set of arc registers are allowed in the system and there is a tool which reads this file which is enough funny in its own little format and spits out C code that is in compiled in with the rest of the system bills what's called a burrs pattern matching which does a optimal for some definition of optimal mapping of Hardware instructions to ideal nodes by doing like an overlapping tree like like usually a hardware node covers two to five ideal nodes and so there's some tiling operation that covers them all including if it eight this region then those edges in between don't have to register allocated but all the edges on the outside do and then

what all the registers that are involved and the registers turn into basically a bit set for every input and output of allowed registers and the register allocator Grox that bit set and uses it to pick the right registers so that gets into registrar in pretty heavily right away because that's how you describe inputs to the register allocator"

Inline Arithmetic functions must be written for each architecture – Arithmetic operations like "+" "-" "*" "/" "mod" "expt" etc. will be translated to a equivalent call e.g. "plus" that can be "inlined" FOR EACH TYPE. Traditionally the instructions of a function have there own place in memory and a jump to that section is performed followed by a jump back to wherever the function was called from. Inline functions mean the whole code of the function is written verbatim each time it is called without any jump instructions. This is the reason programmers have the "inline" primitive, the c code the user feeds to the compiler and if the linker decides to honor the "inline" primitive is where the decision is to be made.

```
// Static Inline function for each possible compare or jump
// for the current architecture in assembly
static inline int foo(){
    return 2;
}
```

At each stage of compilation/translation damcc data will remain in C until there is nothing left but assembly.

"address of" operator: "&" - returns address in memory of a variable "dereference" operator: "*" - Grab data of type of the argument at the given memory location "ffi"-foreign function interface "efi"-external function interface

---

Executable Linkers are basically just home theater setups- Executable Linkers are basically just home theater setups CS361 Chris Kanich https://youtu.be/eQ0KOT_J8Sk?list=PLhy

in certain situations with low memory budgets other techniques might be used – https://github.com/avrdudes/avr-libc

external function interface To allow assembly functions in our compiled code to to be called from another program, we have tor create an external function interface.

https://stackoverflow.com/questions/13901261/calling-assembly-function-from-c

---

1.3 Appendix

Sebastian Falbesoner 2014 Implementing a Global Register Allocator for TCC https://www.complang.tuwien.ac.at/Diplomarbeiten/falbesoner14.pdf

1.4 Events

Probably should write a library for events, hopefully this can be done in a library.

The sublist method of event management, maintains a list of subscribers for each possible event, and when that event is published it is sent out to each of them. Currently this section is in java, but I should translate it into C soon implemented as a C library as well, so we're actually going to describe the operations in java, to avoid any confusion added from also having to juggle pointers and address in addition to the events.

```
import java.util.*;//import of java.util.event

interface ThrowListener { public void Catch(); } //Declaration event inte
// OR import of the interface, OR declared somewhere else in the package

/*_____________________________________________________________*/class Th
//list of catchers & corresponding function to add/remove them in the lis
    List<ThrowListener> listeners = new ArrayList<ThrowListener>();
    public void addThrowListener(ThrowListener toAdd){ listeners.add(toAd
    //Set of functions that Throw Events.
        public void Throw(){ for (ThrowListener hl : listeners) hl.Catch(
            System.out.println("Something thrown");
        }
////Optional: 2 things to send events to a class that is a member of the
. . . go to Thrower.java file see this code . . .
}

/*_____________________________________________________________*/class
implements ThrowListener {//implement added to class
//Set of @Override functions that Catch Events
    @Override public void Catch() {
        System.out.println("I caught something!!");
    }
////Optional: 2 things to receive events from a class that is a member of
. . . go to Catcher.java file see this code . . .
}
```