CSC 431
Final Paper

Alex Gravenor & John Potz

Table of Contents

1 Overview

1.1 Parsing & Internal Representation

The minilang compiler is a pipeline that transforms mini code into a JSON representation, then translates that JSON representation into blz objects that compose an Abstract Syntax Tree. Each object in this Abstract Syntax Tree then represents the kind of node that it represents, and related methods are contained within.

To translate .mini files into JSON, the compiler uses an external tool *ANTLR* to handle lexing and parsing. *ANTLR* is invoked by a bash script that pipes the output into a .json file. This .json file is then opened and read by the main program, which is called from the bash script.

Once the JSON is read from the file and deserialized into a hashmap, it is translated into blz objects using a recursive descent parsing technique. This parser is implemented using the factory pattern. A method *create_node*, looks at the type of the node specified in the JSON and calls an appropriate constructor. Each of these constructors will call *create_node* for each of their child nodes.

1.2 Static Type Checking & Semantics

Type checking is also done recursively, using a store passing style to keep track of the types associated with a binding.

To start the type checking process, first the non primitive types are defined. This means that the structures are added to a map from the type name to their definition. Next, all global bindings are added to a seperate map that keeps track of the currently valid bindings. This includes both global variables and functions. A separate function type is used for the functions.

Then, each function is type checked separately. Each node has a function defined on it that allows for recursive type checking throughout the whole tree. The arguments passed include the current binding map and the list of all of the globally defined types. Each function then returns the updated list of bindings as a result, or throws an error if a type error is found.

To check that every branch of a function returns a valid value, a return equivalence is established. For each kind of node, it may or may not be a return equivalent. For example, the return node is always a return equivalent, but an if node is only return equivalent if both the *then* and *else* blocks are return equivalent.

The return equivalent abstraction allows for checking the existence of certain kinds of unreachable code. Particularly, any code that is in a block following a return equivalent statement. If any such code exists, the compiler will throw an error.

1.3 Intermediate Representation

The compiler uses a control flow graph as an intermediate representation between the Abstract Syntax Tree and the assembly representation. There are several reasons for this intermediate representation. The most important is that it allows for easy analysis of data flow throughout the program. It also allows for more abstract optimizations that may not be visible or straight forward at the assembly level.

Using an intermediate representation, such as LLVM in this case, allows for ignoring particular details and limitations of the assembly language until the last minute. For example, register allocation is extremely tricky to do during code generation. Doing a separate pass is much more straightforward and allows for a more efficient allocation of registers. Single static assignment also enables pushing this register allocation problem to the last minute. If a result goes in a particular SSA register, it is guaranteed that there is no other place that the value could have come from. Thus, we can not worry about accidentally overwriting a particular value in any register.

Additionally, static single assignment allows for easy analysis when attempting to do register allocation. Since a register becomes live when it is assigned to, and is never overwritten, we do not need to track the lifetimes of values, only the register number.

1.4 Optimizations

We implemented several optimizations in our compiler: constant folding, simple sparse constant propagation, control flow graph simplification, and dead code elimination.

1.4.1 Constant Folding

Constant folding is a relatively straightforward optimization. The create_node function used to parse the abstract syntax tree has a multiple return type. It can return a node, or if the expression is constant, it can return a value. Each node that can fold constants (such as addition) checks if all of its children are constant, and if so it will calculate the expression at compile time.

1.4.2 Simple Sparse Constant Propagation

Simple sparse constant propagation is also a relatively straightforward optimization. We setup a worklist as a set which would contain the registers whose uses needed to be evaluated and a hashmap of registers to values with a default value of “top”. Then, we looped through all the registers in the function, set any parameters equal to “bottom” and abstractly evaluated the source for rest of the registers. If the resulting value was anything besides “top”, the register was added to the worklist. Once

all the registers were initialized, we pull registers out of the worklist and abstractly evaluate each instruction that uses the register. If the resulting value is different than what is stored in our hashmap of values for the instruction's destination register, the value is updated and the destination register is added to the worklist. Finally, once the worklist is empty, we loop over all the instructions and for any registers with a constant value get their uses overwritten.

Abstractly evaluating an instruction involves making a guess at the instruction's result without executing the code. As a result, there is limited information to use. Some instructions' results, such as function calls or stores, and some registers, such as function parameters, can only be known at runtime. These registers are assigned a value of "bottom". Occasionally, use of constants will allow the compiler to make some assumptions about the value of a register and these registers will be assigned their constant values. While evaluating the instructions, it is often uncertain if a register's value will be a constant or if it is unable to be determined until runtime. Until a determination is reached, the register is assigned a value of "top". However, once there are no more instructions in the worklist, there should be no registers left with a value of "top". The repeated abstract evaluation of instructions, and subsequent additions to the worklist when a register's value changes, allows the algorithm to propagate constants through the code. The actual abstract evaluation is relatively straightforward, though it differs slightly for different instructions. When any one of an instruction's sources is "bottom", the result is "bottom", and if both of the sources are constants, then the result is a constant calculated based on instruction. In most cases, if one of the sources is "top", then it is assumed nothing yet can be known and the result is "top". However, with phi instructions, we take an optimistic approach and assume, in the case of a single constant value and one or more "top" values, the result is the constant value. However, if there are multiple constant sources, or any "bottom" sources, the result is "bottom".

Using SSA was a significant help with this optimization. It allowed us to easily track the source instruction for each register, ensuring a single source for each register. Additionally, we were easily able to create a list of all the instructions using the register.

1.4.3 Control Flow Graph Simplification

For control flow graph simplification, we implement three different sub optimizations. They are, transforming constant conditional branches into unconditional branches, removing unconditional branches, and removing empty blocks.

When a conditional branch has an always true or always false condition, it can be translated into an unconditional branch into the corresponding block. Additionally, when a conditional branch has the same destination for both the true and false case, it can be translated into an unconditional branch into that block. These optimizations cut

down on the total number of conditional branches in the program, which are bad for performance because they may cause cache misses and speculative execution flushes.

An unconditional branch can always be removed by combining the source and destination blocks. This cuts down on the total number of instructions in the program, and moves code closer together, which improves locality of code. Improving locality of code cuts down on the number of cache misses during execution, which in turn improves speed.

Similarly, empty blocks serve no purpose, but may cause cache misses. Thus, removing them has no impact on the actual behavior of the program, but may help performance.

1.4.4 Dead Code Elimination

To eliminate dead code, the compiler will compute which instructions are used. This is done by defining a core set of instructions that may have a meaningful impact on the computational state of the program. This set includes things such as returns, function calls, and stores. Because the compiler cannot be sure of what side effects these have, it must assume that the instructions are necessary. If the instructions are necessary, their sources must also be necessary, since the registers have a single static assignment, there is exactly one instruction that assigns to that register. That instruction is marked as necessary, and the process is repeated until there are no more unprocessed necessary instructions. Any instruction not marked as necessary is removed as dead code.

Each of these optimizations can enable opportunities for other optimization to do more, so the best result occurs when all optimizations are enabled.

1.5 Code Generation & Register Allocation

When translating from LLVM to assembly, the llvm registers must be replaced with registers that physically exist on the machine. This is done by a register allocation algorithm that creates a graph of which llvm registers are live at the same time. This graph is then colored using a graph coloring heuristic. If a register is not found for the register, then it is considered a *spilled* register. Space is allocated on the stack for the value of this register to be stored. Two additional registers are reserved for reading from and storing to these spilled spaces.

Most LLVM instructions are direct translations to assembly instructions, however the phi instructions are an exception. No concept of a phi exists. Instead, we must change phi instructions into *mov* instructions. For each phi instruction, we place a *mov* instruction into the block where the value comes from, directly preceding the branch. This could not be done at the LLVM level, because this violates static single

assignment. This change lets the right value be in the same register no matter which branch execution comes from.

1.6 Other

One great performance improvement that this compiler has is that it is able to run in multiple threads. After the initial parsing and type checking phases, there is no need for any function to know anything about any other function. Thus, the code generation and optimization of each function can be done in separate threads. To implement this, a single coordinator thread spawns a thread for each function and waits for it to return with the compiled version of that function. Once the master thread has detected that every function thread has produced output, it will exit the program.

This performance trick was definitely necessary because the compiler was written in blz, which is an interpreted language. Therefore it is quite slow at some of the more computationally heavy operations such as register allocation and constant propagation. However, blz does grant some unique flexibility in writing the code of the compiler. Being a dynamically typed language, return types of functions can take on many types. This was used heavily in the parsing of the language to either return AST nodes or constant values that were folded up. Another heavily used feature of blz was mapping or filtering over an array with a lambda. This allows concise code to replace several lines of a loop that ends up doing something simple.

2 Benchmark Analysis

2.1 Methodology

Our compiler produces two classes of output, LLVM intermediate representation and ARM32 assembly. Because of the setup on our ARM32 machine, we were unable to compile the LLVM to run on that machine. Thus, we have two result sets.

One set is compiled ARM32 assembly. This is compared to a clang result of comparable C code. This is all run on a raspberry pi. These were run with the shorter input for each benchmark.

The other set is LLVM intermediate representation that is compiled to x86_64 by clang without any further optimizations. These tests were run on a modern laptop. These were run with the longer input for each benchmark.

2.2 Limitations

Some benchmarks took too long to compile with certain settings, brett and OptimizationBenchmark both took more than 6 hours to compile to assembly with and without optimizations. Thus, it was impractical to actually compile these programs. Additionally, a few of the programs compiled incorrectly to assembly, and these programs segfault. This means that timing them was not possible.

2.3 Results

Runtime results are graphed in Fig 2.3.1-4. Some results are tricky to see on the graph because the values are so small that they are illegible compared to the other results, so please refer to the table in Fig 2.3.5-6 to see raw averages.

Looking at how the compiler performed on ARM assembly output, the base compiler actually achieves great performance compared to clang. Compiling without optimization often matches or outperforms the clang output. However, the optimizations appear to not be very effective in improving assembly performance. Many of the optimized outputs actually behave slower or the same as their unoptimized counterparts. Because the runtimes of these programs are so short, it is likely that the error in measurement may mean that most of these results are not statistically different.

The LLVM output yields far more varying results. For these benchmarks, optimizations made improvements in runtime speed in almost all cases. However, clang with optimization level 3 managed to beat our compiler in every test case except for *stats, biggest,* and *bert*.

Fig 2.3.1

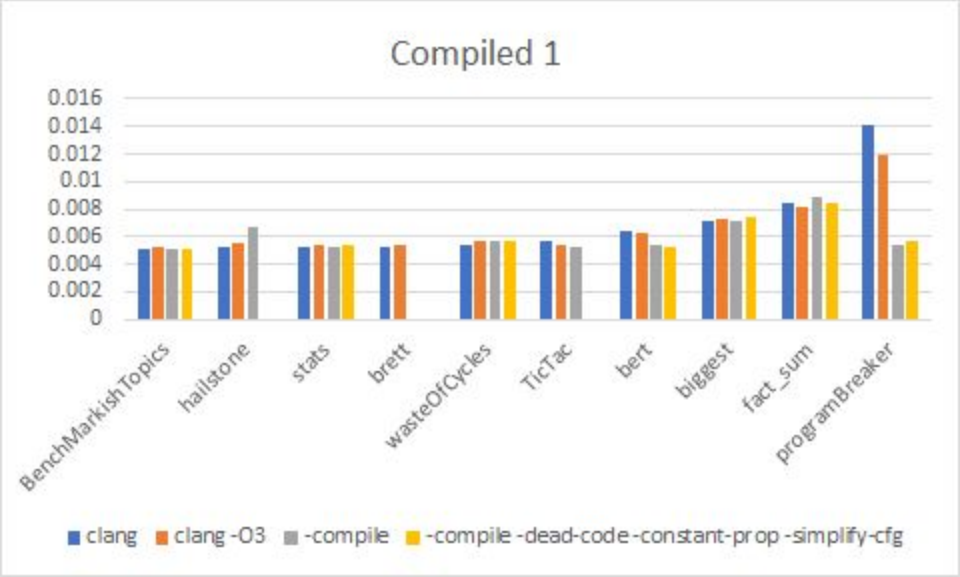

Fig 2.3.2

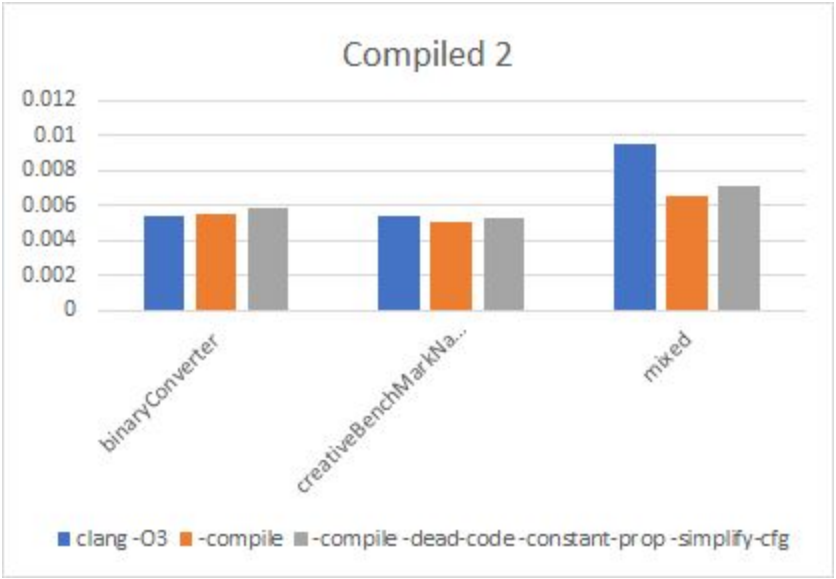

Fig 2.3.3

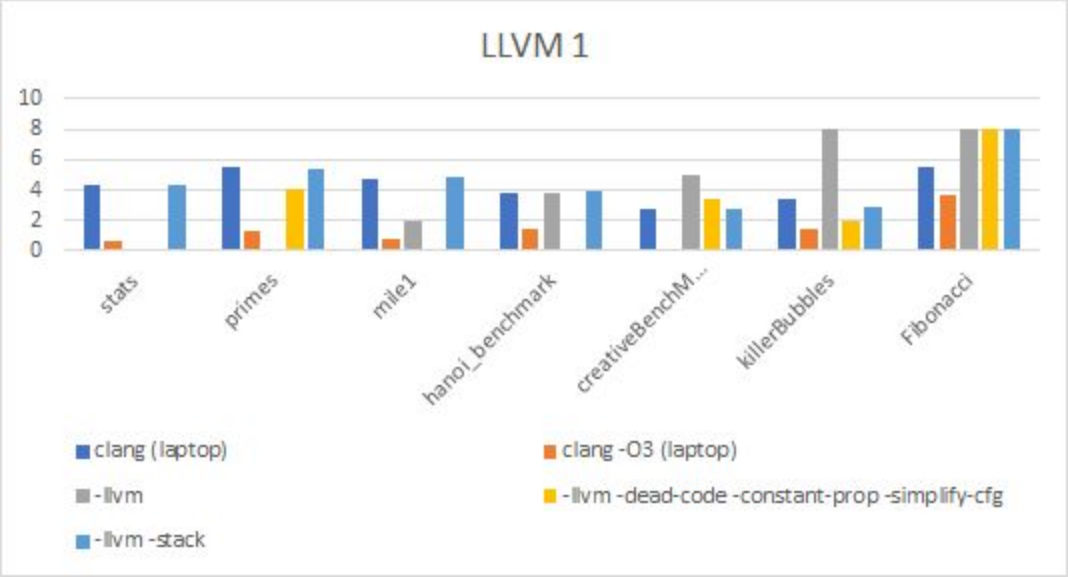

Fig 2.3.4

Fig 2.3.5 (Assembly Average Run Times)

Program	clang	clang -O3	-compile	-compile -dead-code -constant-prop -simplify-cfg
BenchMarkishTopics	0.00508	0.005263	0.005079	0.005163348
hailstone	0.005275	0.005491	0.006757	ERR
stats	0.005286	0.005399	0.005188	0.005423331
brett	0.005289	0.005371	DNC	DNC
wasteOfCycles	0.005349	0.005623	0.005719	0.00567714
TicTac	0.005707	0.005402	0.005209	DNC
bert	0.006405	0.006342	0.00539	0.00520165
biggest	0.007075	0.00736	0.007122	0.00743345
fact_sum	0.008491	0.008141	0.008894	0.008488488
programBreaker	0.014169	0.01199	0.005432	0.005673003
mile1	0.092946	0.021507	DNC	0.005558813
OptimizationBenchmark	0.126911	0.005359	DNC	DNC
primes	0.992604	0.448886	ERR	ERR
hanoi_benchmark	1.936124	0.933718	0.005163	ERR
GeneralFunctAndOptimize	4.542559	1.095748	SF	SF
binaryConverter	7.376303	0.005446	0.005488	0.005888617
creativeBenchMarkName	8.527412	0.005452	0.005117	0.005276012
mixed	9.044881	0.009568	0.006605	0.007136989
Fibonacci	12.15061	6.957941	14.78704	14.81373213
killerBubbles	15.83536	4.44927	SF	SF

Key: ERR - program took too long to execute;
DNC - program took too long to compile;
SF - program caused a seg fault when run

Fig 2.3.6 (LLVM Average Run Times)

Program	clang (laptop)	clang -O3 (laptop)	-llvm	-llvm -dead-code -constant-prop -simplify-cfg	-llvm -stack
brett	0.00151391	0.001615858	0.001728735	0.00300046	0.002395956
TicTac	0.001756275	0.002021658	0.001832699	7.126294386	0.002207392
fact_sum	0.002453303	0.002425408	0.002072949	1.969831065	0.00193877
stats	4.335397589	0.6506127	0.002245389	0.001765776	4.31708873
biggest	0.002135408	0.002358687	0.002318759	0.001851546	0.001924264
bert	0.002040708	0.001693344	0.002596541	0.001452684	0.002201708
primes	5.504387891	1.334204912	0.003403413	4.008460939	5.33069657
GeneralFunctAndOptimize	0.634048092	0.131419218	0.575845191	0.00396977	0.568666709
wasteOfCycles	0.001401281	0.002272379	0.57641834	0.645857203	0.001242161
mile1	4.785000432	0.826906705	1.9642469	0.00389967	4.793050866
hanoi_benchmark	3.74446131	1.405674922	3.860233809	0.001445949	3.892586934
hailstone	0.001612663	0.001848769	3.957645027	0.001769342	0.003125404
creativeBenchMarkName	2.795438695	0.001899517	4.967531995	3.357076294	2.703317065
programBreaker	0.002927589	0.003331256	6.259595118	0.001821864	0.003315072
OptimizationBenchmark	6.196144104	0.004061985	7.66078634	0.002317464	6.245218227
killerBubbles	3.402875626	1.482643044	8.039181483	1.969105804	2.871268486
Fibonacci	5.576284695	3.617829454	8.046242726	8.038703667	8.001545153
BenchMarkishTopics	13.036384	6.209022594	9.804821529	0.002367609	9.897443947
mixed	2.861284637	0.862625527	12.72977271	0.001849234	2.101880136
binaryConverter	9.853974771	0.00127877	16.66293816	0.001855813	9.825399826