Shayna Goldfarb

Professor Xi

CS 391

25 June 2025

Lambda Calculus Compiler Explained

This report documents the design and implementation of a compiler for an extended lambda calculus language that targets Python. The compiler translates high-level terms that represent lambda expressions into a low-level intermediate representation (IR), which is then converted into executable Python code.

The compiler’s primary components include:

* A type checker to validate terms
* A compiler that emits an intermediate representation (temp)
* A code emitter that translates IR into Python code

The first type of high-level language is “term”. This data type represents an expression and supports many different constructs. For example, it supports constants (integers and Booleans), variables, lambda abstraction, applications, operators (such as addition, multiplication, not equal, etc.), if-else expressions, fixed-point functions, tuples, and finally let-bindings.

For example, if you wanted to represent the operation “3+4” using terms, it would be term\_add(term\_int(3), term\_int(4)). term\_add is already defined in the code as term\_add = lambda a1, a2: term\_opr("+", [a1, a2]). term\_opr takes in a string representing the operation and a list of terms. term\_int represents an integer. All together, 3 and 4 are represented by term\_int constants, term\_opr takes in those two constants as a list and also takes in the plus sign as an addition operation, and term\_add uses the word “lambda” to define a function.

Another data type is “styp”. This data type is short for static type and is used to represent types of terms during type checking. styp\_bas represents a string that declares a type (such as “int”, “bool”, etc). styp\_tup represents a tuple of two styp values while styp\_fun represents a function of two styp values. The function styp\_equal checks if two styp values are the same type.

While term represents an expression, styp represents a type that describes what kind of value a term produces. For example, term\_int(3) represents the number 3, while styp\_bas(“int”) represents the type integer.

The next data type is “tctx” which represents a typing context/environment. This type keeps track of variables and their types using tctx\_nil (representing the empty context) and tctx\_cons (representing a context with a variable name, type, and the rest of the environment).

This data type maps a variable name to a type. It combines the two previous types, term and styp, to typecheck a term. For example, a term is term\_int(3), a styp is styp\_bas(“int”), and a tctx is tctx\_cons(“3”, styp\_bas(“int”), tctx\_nil()).

After defining tctx\_cons, the code implements a type-checking function called term\_tpck01 which traverses a given term and uses the typing context to look up variable types and ensures consistency. For example, print(term\_tpck00(term\_add(int\_1, int\_1))) returns styp\_int, making sure that the addition function returns an integer. The function term\_tpck00 takes in a term and calls term\_tpck01 with that term and a null context.

This function also calls another function called tctx\_search, which takes in a tctx data type and an argument. If the value of the variable in the context matches the argument passed through the function, it returns the type in the context. If it doesn’t match, then it recursively calls itself on the rest of the arguments. For example, in the variable branch of term\_tpck01, tctx\_search is used to make sure that variable is in the context.

After that is the intermediate representation (IR). The first is tval which represents a value/constant in the IR. It can either be tval\_int (an integer), tval\_btf (a Boolean), or tval\_reg (a registered reference).

The next IR data type is treg, a temporary register that stores the result of a computation. Registers are uniquely named using prefixes (e.g., tmp, arg, fun) and a counter. After a term is compiled, its value is stored in a treg. The targ\_new function creates a treg with the prefix “arg” and the integer 0, holding the input argument for a function. The ttmp\_new function creates a treg with the prefix “tmp” and a unique value, holding the intermediate results during computation. It’s usually used when evaluating subexpressions. The tfun\_new function creates a treg with the prefix “tfun” and a unique value, holding the compiled lambda expression.

The next IR data type is tins, an instruction. Tins contains different types of instructions (such as operations, if-else statements, and applications) represented as tins\_mov(dst, src), tins\_opr(res, opr, [r1, r2]), tins\_app(res, fun, arg), tins\_if0(res, test, then, else), and tins\_fun(f00, body). Tins\_mov moves a source (integer tval\_int) into a destination (temporary register treg). Tins\_opr performs an operation on two values (r1 and r2) and places the result into a treg. Tins\_app applies a function fun onto a treg (arg) and places the result into a treg (res). Tins\_if0 represents an if-then-else statement where the if test is represented by a treg (test), the then and else results are represented by two tcmps, and the result of the statement is stored in a treg (res). Tins\_fun defines a treg function with a tcmp representing the body of the function.

Tval is the actual value, treg is where the value is stored, and Tins represents actual operations. In Python we can write the code x=3. In the compiler, treg is the variable x, tval is the integer 3, and tins is the entire line of code. Using the IR data types, ttmp=3 would be represented as tins\_mov(treg(“ttmp”, 101), tval\_int(3)).

Another IR data type is tcmp, a computation. This includes a list of instructions (tins) and the final result register. In Python, a tcmp represents an entire function.

Then there is code registration. The last IR data type is compiler environment (cenv): cenv\_nil (empty environment) and cenv\_cons (bind variable to a register). Cenv keeps track of different variables’ values during compilation. The function cenv\_search looks though the environment to find a treg whose value matches the input value x00.

Cenv and tctx can look very similar at first glance. However, they are used in two different parts of compilation. Tctx checks types by cross-checking variables to their types (styp). Cenv generates code by mapping variable names to their registers (treg).

The function term\_comp01 translates a term into IR. It generates new temporary registers, emits appropriate instructions, and manages compiler environments. It takes in a term and a compiler environment and returns a computation (tcmp). If the term is an integer, term\_comp01 creates a new treg ttmp, moves that integer value into ttmp and returns the tcmp of the tins\_mov instruction and ttmp. The function operates the same way for Booleans. For a variable, the function uses cenv\_search to find a register with that variable name and returns a tcmp with an empty instruction list and the name of the register with the variable. For an operation, the function recursively calls itself for each value, creating a tcmp for each one. It then generates a list of instructions containing the two lists of instructions (from each tcmp) and stores the value of tins\_opr on the two values and a new ttmp as a list entry. It then returns a new treg and that list of instructions. The application, if-else, tuple, fst, snd, and let branches work similarly as they all call term\_comp01 recursively on each term, create a giant list of instructions for each term and the final instruction (e.g. application or if0), and return a tcmp of that list and a register.

There are some translation rules. For example, the constants create a mov instruction. Application evaluates function and argument, then emit app. The if-expression emits if0 with both branches. The lambda generate a function instruction.

The final part of the code is the Python code emission. The function tcmp\_pyemit converts a tcmp object into a readable and runnable Python function. It processes each instruction, emits valid Python syntax, and returns the final result register. The mov branch returns a direct assignment, while the opr branch returns an infix operation. The application branch returns a function call while the function branch defines nested Python functions. Lastly, the if-else branch emits if/else branches.

Tcmp\_pyemit starts by printing “def main():” and then defining functions to emit code. Emit\_val(tv) checks to see if tv is a tval, and return its value as a string if it is. Emit\_ins(ins) returns what to print for each type of instruction. For example, if ins is a tins\_mov, then it will print the register name and the value. If ins is a tins\_app, it converts each ins argument into a string and prints them as readable Python code. For tins\_fun, the function calls itself recursively to turn each line of code in the function body into readable Python code. After defining both functions, tcmp\_pyemit runs emit\_ins for each instruction in the tcmp object. At the end, it prints the return statement to the result register.

For example, applying term\_dbl to the tcmp\_pyemit function returns

“def main():

def fun101(arg0):

tmp102 = arg0 + arg0

return tmp102

return fun101”

term\_dbl is defined earlier in the code as a function that doubles an integer – shown as term\_dbl = term\_lam("x", styp\_int, term\_add(var\_x, var\_x)). In this representation, the tcmp\_pyemit function takes in the high-level terms (such as term\_lam and term\_add) and returns a function written in Python that takes one integer, doubles it, and returns the result of that addition.

In more detail, it would go through the TMlam case of term\_comp01: assign x01=”x”, creates fun0=treg(“fun”, 101), creates arg0=treg(“arg”, 0) and creates cenv\_cons of x01, arg0, and cenv\_null. Then it recursively calls term\_comp01 on term\_add, which is a term\_opr for adding two values. Looking at the TMopr case, it also recursively calls itself for each value to find the tcmps. Then TMopr returns a tcmp of one instruction (tins\_opr on the registers) and puts the result on register ttmp, which is assigned to cmp1 in the TMlam case. Next it calls tins\_fun on fun0, cmp1, and arg0 and stores the result to inss as a list. Finally, it returns a tcmp with tins\_fun as an instruction and fun0 as the result register. Therefore the final value should be tcmp([tins\_fun(fun101, [tins\_opr(tmp102, "+", [arg0, arg0])])], fun101).

The final system is capable of compiling and executing programs such as factorials and the 8-queen puzzle problem.