Veer Agrawal
U25447882
CAS CS391
June 24th

Final Report

Introduction

This project implements a compiler that translates a functional language—an extended version of the lambda calculus—into Python code. The compiler is built directly on the code and structure provided in lecture-06-18/lambda3.py and enhances it with support for additional language constructs, including tuples, projections, let-bindings, and extended operators.

The overall project is split into two major tasks:

- **Task 1:** Extend the compiler's core to support new language constructs, including tuples, projections, let-bindings, and binary operators.
- **Task 2:** Implement a Python code emitter that transforms the compiled intermediate representation into runnable Python functions.

This report follows the structure of the compiler itself, emphasising key data types, core functions, and the transformation pipeline from high-level LAMBDA code to Python output.

Language Design: Data Types and Syntax

Functional Terms

The source language for this compiler is a variant of the lambda calculus as introduced in class. It supports a core set of functional constructs, each modelled as a subclass of a base term class. These constructors were already provided in the professor's code:

term_int: integer literals, **term_btf**: boolean literals, **term_var**: named variables, **term_lam**: lambda abstractions (functions), **term_app**: function applications, **term_fix**: fixed-point recursion for defining recursive functions, **term_if0**: conditional branching based on whether a boolean is true or false, **term_opr**: primitive operations like addition, multiplication, comparison, etc.

Additional Extensions (Implemented in Task 1)

To expand the language's expressiveness and enable more complex programs (such as the 8-Queens solver), I implemented support for the following additional term constructs:

- **term_tup(t1, t2)**: Represents a tuple or pair of values.

- **term_fst(t)**: Extracts the first element of a tuple.
- **term_snd(t)**: Extracts the second element of a tuple.
- **term_let(x, t1, t2)**: Represents a let-binding, where the result of evaluating t1 is bound to the variable x in the scope of t2.

Compilation to Stack-IR (Task 1)

Function: term_comp01(tm0, cenv)

This function compiles a term tm0 into a stack-based IR called tcmp. It recursively walks through the term syntax tree, accumulating a list of instructions (tins) and returning a final register containing the result.

Binary Operator Handling

For all 11 operators (+, -, *, <, >, <=, >=, =, !=, cmp), I emit TINSopr instructions using a helper rule:

```
cmp1 = term_comp01(ags[0], cenv)
cmp2 = term_comp01(ags[1], cenv)
ttmp = ttmp_new()
inss = cmp1.arg1 + cmp2.arg1 + [tins_opr(ttmp, pnm, [cmp1.arg2, cmp2.arg2])]
```

This pattern compiles binary expressions into an IR where both operands are evaluated, and the result is stored in a temporary register.

Tuple Construction and Projection

The tuple-related constructs (term_tup, term_fst, term_snd) extend the functional language to allow for structured data and decomposition of values.

- **term_tup(t1, t2)**
  This construct represents a pair of terms. During compilation, both t1 and t2 are first recursively compiled into their corresponding computations, producing result registers r1 and r2. A new temporary register is then allocated, and the compiler emits a tins_opr(tmp, "tup", [r1, r2]) instruction. This instruction models tuple construction in the Stack-IR.

- **term_fst(t)**
  To extract the first element of a tuple, I first compile t to obtain a register reg holding the tuple value. I then allocate a new temporary register and emit tins_opr(tmp, "fst", [reg]). This Stack-IR instruction models tuple projection and ensures that the tuple remains untouched while extracting its first component.

- **term_snd(t)**
  The process mirrors that of term_fst: the term t is compiled to a register holding a tuple, and then tins_opr(tmp, "snd", [reg]) is emitted to access the second component.

Let-Bindings and Lexical Scoping

term_let(x, t1, t2): This enables naming and reuse of intermediate results. The compilation proceeds in three structured steps:

1. **Compile t1**: The expression t1 is recursively compiled into a computation that returns a register r1.
2. **Extend the environment**: The compiler introduces a new binding into the compilation environment (cenv) by associating variable name x with register r1. This extension is done by creating a new cenv_cons entry.
3. **Compile t2 in the extended environment**: Now that x is bound to r1, we recursively compile t2. Any occurrence of x in t2 is resolved through cenv_search, which respects lexical scoping and will return r1 for x.

The use of the environment ensures that variable bindings are well-defined and that nested scopes are handled cleanly, with each variable mapped to the correct temporary register at compile time.

Stack-IR Data Model

The intermediate representation (IR) of the compiler is a low-level stack-oriented instruction language designed to capture the operational semantics of lambda expressions. This representation consists of several structured components:

Registers (treg)

Registers are abstract placeholders used to store intermediate results during computation. The compiler employs helper functions like ttmp_new(), targ_new(), and tfun_new() to ensure unique and consistent naming throughout the compilation process.

Values (tval)

Literal constants in the IR are represented using the tval type. These include TVALint(n): an integer literal n, TVALbtf(b): a boolean literal True or False. These values serve as immediate constants in the emitted instructions.

Instructions (tins)

Instructions represent atomic steps in the computation. The IR supports the following instruction types:

- TINSmov: Assigns a constant value to a register (e.g., tmp101 = 5)
- TINSopr: Applies a binary operator or tuple operation to operands (e.g., tmp103 = tmp1 + tmp2, or tmp105 = (tmp1, tmp2))
- TINSapp: Represents function application by calling a function register with an argument
- TINSfun: Defines a new function, taking a parameter register and a body computation
- TINSif0: Encodes conditional branching logic based on a test register

Each instruction is a structured object, and the order in which instructions are listed encodes the sequence of evaluation.

Computations (tcmp)

A computation is represented by a tcmp object, which consists of: A list of instructions (tins), to be executed in order and a final result register, which holds the outcome of the computation

Environment (cenv)

cenv_cons("x", reg_x, cenv_cons("y", reg_y, cenv_nil()))

This structure maps variable names to their corresponding result registers. When compiling variable references (TMvar), the compiler performs a lookup using cenv_search, which walks the environment chain to find the most recent binding. This design ensures that variables are correctly resolved at each level of the function hierarchy.

Code Emission: Python Translation (Task 2)

Once a high-level lambda term is compiled into a low-level computation (tcmp), the final step is to **emit executable Python code**. This is accomplished through the tcmp_pyemit function, which walks the internal stack-IR and translates each instruction into valid Python syntax.

Overview of tcmp_pyemit

The function tcmp_pyemit(cmp0, fname) takes a compiled computation cmp0 and emits a complete Python function definition named fname. The generated code mirrors the instruction sequence and control flow encoded in the computation.

The emission logic is handled recursively via the helper function tcmp_pyemit_cmp(cmp0, indent), which returns a list of Python lines and the final result register.

Register and Value Conversion

Two helper functions convert intermediate representations into Python-friendly syntax:

- treg_py(treg0): Converts a register object like treg("tmp", 101) into a Python variable name, such as "tmp101". This ensures a clean, readable mapping from the IR to

concrete Python identifiers.

- tval_py(tval0): Converts values in the IR to Python literals. For instance:
    - TVALint(5) → "5", TVALbtf(True) → "True"

The function tcmp_pyemit_cmp recursively traverses the compiled intermediate representation (IR) and emits equivalent Python code. Each instruction variant in the IR maps directly to a corresponding Python construct:

- **TINSmov**: Assigns a literal value to a temporary: tmp101 = 42
- **TINSopr**: Handles binary operations, tuples, and projections: tmp = a + b (arithmetic), tmp = (a, b) ("tup"), etc
- **TINSapp**: Applies a function to an argument: tmp = fun(arg)
- **TINSfun**: Defines a Python function with a parameter and a return.
- **TINSif0**: Translates to an if/else block with proper indentation.

The top-level tcmp_pyemit wraps this logic into a complete Python function. For example, compiling the 8-Queens lambda term yields a full Python function eight_queen_solver() that, when executed, returns the correct result: 92.

Testing

To verify correctness, I conducted several tests.

1) 8-Queens Puzzle

I compiled the full lambda-calculus solution to the 8-queens puzzle (written in midterm.py) using my final compiler. The compiled output was emitted as Python code and then executed. The solver correctly returned the emitted code and also the generated solver output: 92. This confirmed that the full pipeline is working.

2) Unit Tests: Compiler Features

I also tested all of the core compilation features added in Tasks 1 and 2:

- **Binary Operators**: Verified correct tcmp generation for all 11 operators: +, -, *, <, <=, >, >=, =, !=, cmp.

- **Tuples & Projections**:
    - Construction: term_tup(term_int(1), term_int(2))
    - Access: term_fst(tup), term_snd(tup)
- **Let Bindings**:
    - Example: term_let("x", term_int(42), term_var("x"))
- **Lambda + Application**:
    - (λx. x + 1)(2)

- **Nested Expressions**:
    - let p = (4,5) in let x = fst(p) in let y = snd(p) in x + y

Each expression was compiled to stack-based IR (tcmp) without error and matched the expected behaviour when emitted as Python. Together, these tests show that the compiler handles all required constructs and extensions and is capable of translating real-world recursive functional programs.