LILA – LLM-based Intelligent LLVM Assistant per la generazione e debugging di programmi in Grammo

Mario Cosenza
NF22500094

Contesto e obiettivi

LILA è un sistema multi-agente basato su LLM che assiste l'utente nel ciclo completo di sviluppo di programmi in **Grammo**, un linguaggio ideato da **Salvatore Di Martino**, nel contesto del corso di Ingegneria dei Linguaggi di Programmazione.
Dato il compilatore per Grammo in grado di produrre **LLVM IR**, LILA considera questo compilatore come un "tool esterno" e si concentra sul livello superiore di orchestrazione: interpretazione dei requisiti, pianificazione, generazione di codice, gestione degli errori di compilazione e correzione iterativa.
Obiettivo principale del progetto è progettare e realizzare un'architettura di **agenti LLM specializzati**, coordinati tramite un orchestratore, che sappiano usare in modo robusto i tool disponibili e che possano essere valutati con metriche adeguate al contesto di code generation.
Tutto deve funzionare senza API a pagamento, sfruttando **modelli open-source eseguiti in locale**, con la possibilità di utilizzare in maniera limitata un modello esterno accessibile in modalità gratuita (ad esempio Gemini).

Architettura di LILA: agenti e flusso di esecuzione

L'architettura si basa su un set di agenti LLM specializzati, orchestrati da un grafo di esecuzione. Ogni agente ha un ruolo preciso e un insieme di tool che può invocare.

- Al centro c'è l'**Orchestrator**, unico punto di contatto con l'utente. L'Orchestrator riceve una descrizione del problema, decide il percorso nel grafo: prima il Planner, poi il Code Generator, successivamente il Compiler Interface e infine il Debugger/Evaluator in caso di problemi.
- Il **Planner** è l'agente incaricato di trasformare il requisito in un piano tecnico. A partire dal prompt iniziale, il Planner individua come suddividere il problema in sottoproblemi. Il risultato del Planner non è codice, ma una struttura che descrive "cosa" va implementato e con quali vincoli.
- Il **Generator** prende in input il piano del Planner e si occupa di produrre effettivamente il sorgente in Grammo. Qui il ruolo del LLM è centrale: deve rispettare la sintassi, le regole di scope e di tipo, e allo stesso tempo tradurre in codice le decisioni di progettazione prese dal Planner. Il Generator ha tool per eseguire un controllo sintattico tramite il parser del linguaggio, costruito in Lark. In ogni iterazione genera una versione candidata del programma, verifica almeno la correttezza sintattica con il parser e, se non emergono errori immediati, chiede all'Orchestrator di passare il controllo all'agente successivo.

- La **CompilerInterface** è lo strato che collega LILA al compilatore vero e proprio. Non è solo un "wrapper" banale: il suo compito è esporre al resto del sistema tool robusti con interfacce ben definite, del tipo "compila questo sorgente e restituisci esito, IR e messaggi d'errore strutturati" oppure "esegui il programma compilato con questi input e restituisci output e log". L'obiettivo è fornire a Orchestrator, Generator e Debugger un modo uniforme di interagire con il compilatore, nascondendo i dettagli di basso livello.
- Quando la compilazione fallisce e test non passano, entra in gioco il **DebuggerEvaluator**. Questo agente riceve una descrizione strutturata dei problemi: errori di compilazione (tipo, posizione, messaggio). Deve interpretare questi segnali alla luce delle regole di Grammo e produrre due output: una spiegazione comprensibile dell'errore e una proposta di correzione. Le correzioni vengono rappresentate come patch al codice o aggiornamenti al piano del Planner: ad esempio "aggiungere un controllo sul caso lista vuota prima di accedere all'indice 0", "spostare l'inizializzazione di una variabile all'inizio della funzione", "correggere il tipo di ritorno di una funzione per allinearlo ai test".

Il flusso tipico è quindi: Orchestrator riceve la richiesta, il Planner costruisce il piano, il Generator produce il codice, il CompilerInterface compila ed eventualmente esegue, il DebuggerEvaluator analizza gli errori e suggerisce correzioni.
La presenza di ruoli specializzati permette di controllare in modo più fine come l'LLM interagisce con i tool e dove si generano gli errori.

Tecnologie e librerie

LILA è pensato per essere realizzato interamente in **Python**. L'orchestrazione del flusso e lo stato globale della "sessione di compilazione assistita" sono gestiti da **LangGraph**, che permette di definire il grafo degli agenti e le condizioni di transizione tra uno stato e l'altro. **LangChain** fornisce la definizione degli agenti con tool: ogni agente è un modello LLM con un prompt specifico e una serie di tool compilatore, parser, analizzatore di errori.
I modelli LLM vengono eseguiti in locale su RTX 3070, con un server **Ollama**, che espone i modelli via HTTP o in alternativa è tramite l'API gratuita di Gemini.
Il compilatore è incapsulato in Python ed offre operazioni quali: compila un file, restituisci l'IR, prova ad eseguire con determinati input.

Metriche per valutare LILA e il modello LLM

La valutazione del progetto riguarda il comportamento del sistema LILA e misura quanto bene l'LLM, tramite agenti specializzati, sa usare i tool e generare programmi corretti. Per ogni task si osserva anzitutto la correttezza del codice prodotto: si calcola quante soluzioni compilano al primo tentativo e le metriche **Pass@k**, che indicano la capacità del sistema di risolvere il problema entro k cicli di generazione e debug.