Andrea Simone Costa - 597287

Scanner

Lo scanner non presenta particolari differenze rispetto a quanto visto a lezione. Lo scanner dichiara una serie di espressioni regolari di varia utilità, ad esempio per riconoscere numeri in base decimale e binaria, e una hashmap per il riconoscimento delle keyword. Sono definite tre regole: una per riconoscere il token successivo, una per gestire i commenti su linea singola e la terza per gestire i commenti multilinea.

Parser

Il parser utilizza le API monolitiche di Menhir, dichiara tutti i token necessari e fa estensivo uso della precedenza assegnabile ad essi per risolvere la maggior parte dei conflitti tra regole, come ad esempio il classico problema degli if-then. Per risolvere altre situazioni conflittuali è stato fatto uso del modificatore `%inline` oppure di una appropriata riscrittura delle produzioni, non sempre intuitiva, per evitare le ambiguità.

Tra le definizioni ausiliarie troviamo:

- `vardesc_typ`, un tipo che dichiara dei costruttori base per tenere traccia delle varie star, parentesi tonde e quadre incontrate durante il parsing di una `vardesc`
- `from_vardesc_to_ast_type`, una funzione in grado di trasformare una lista di `vardesc_typ` in un tipo ben formato inseribile nell'AST
- `|@|`, una funzione per aggiungere al volo la location a un nodo creando un record

Poiché vi è il supporto a la dichiarazione di variabili con inizializzazione opzionale e dichiarazioni multiple è stata creata una produzione apposita, `vardecl_init_list`, che si cura di generare l'AST corrispondente dopo aver costruito il tipo di ogni dichiarazione con `from_vardesc_to_ast_type`.

Il supporto al comma operator rasenta quello del linguaggio C ma è leggermente meno potente. È stata inserita una produzione apposita, `expr_comma`, che lo contiene come sotto-produzione, a differenza della classica `expr`. Il problema che si è voluto risolvere creando questa distinzione riguardava i conflitti sorti con l'invocazione di funzioni e la lista di inizializzazione degli array, luoghi nei quali compaiono espressioni intervallate da virgole. In queste situazioni è stata imposta la produzione `expr`, che non contiene il comma operator tra le alternative immediate, mentre in tutti gli altri luoghi viene fatto uso della produzione `expr_comma`.

Il parser riconosce i cicli `for` e il ciclo `do-while`, ma effettua una operazione di riscrittura di questi ultimi nel ciclo `while`. Altre operazioni di riscrittura sono state eseguite per gli operatori di pre- e post-decremento, tramite il comma operator, e per le abbreviazioni degli assignment operator, che vengono espanse.

Poiché il linguaggio non ammette funzioni che restituiscono array o puntatori, tale invariante è stata imposta nella produzione per la dichiarazione di funzione.

AST

L'AST fornitoci ha subito le seguenti modifiche:

1. è stato aggiunto il costruttore `Null`, un `expr_node` apposito per il `NULL`
2. è stato aggiunto il costruttore `Comma`, un `expr_node` dedicato al comma operator che contiene una lista di espressioni
3. sono stati modificati i costruttori `topdecl_node.Vardec` e `stmtordec_node.Dec` per supportare la dichiarazione di variabili multiple con corrispondente inizializzazione

È stata definita anche una funzione ricorsiva per trasformare un AST in una stringa utilizzando una "notazione" minimale, per facilitare la lettura e la comprensione di un AST stampato a video.

Symbol Table

La symbol table è una lista immutabile di hashmap mutabili, una per ogni scope incontrato. Questa configurazione si è rivelata il compromesso ideale: all'inizio di ogni blocco o di ogni funzione è possibile agilmente aggiungere una entry alla lista senza intaccare gli scope esistenti fino a quel momento, le modifiche ad ogni hashmap rimangono visibili per tutta l'esistenza dello scope e all'uscita in automatico esso viene deallocato, in quanto viene meno il riferimento alla lista che lo contiene.

È stata definita anche la funzione `print_keys` per stampare gli identificatori presenti in ogni livello della lista.

Semantic analysis

Nel file di analisi semantica troviamo l'implementazione del type system. Vengono definiti i tipi delle entità manipolate dal linguaggio e una utility di conversione dalle annotazioni di tipo presenti sull'AST. È interessante notare che il tipo delle funzioni è soggetto a currying. Ad esempio un tipo funzione come `int f(int a, bool b)` diventa `int -> bool -> int`, mentre un tipo funzione come `int f()` diventa `void -> int`. Poter trattare un parametro alla volta si rivela molto agevole in fase di type checking delle chiamate a funzione.

La relazione di equivalenza tra tipi, `check_type_equality`, è un controllo ricorsivo di assegnabilità in realtà che fa perno sulla stretta uguaglianza tra i tipi in gioco, poiché non è prevista una nozione di sottotipo. Per quanto riguarda gli array, si ha che due array non sono mai l'uno assegnabile all'altro, tranne nel caso di chiamata a funzione: l'assegnabilità è consentita solo se entrambi dichiarano la medesima size oppure se almeno uno dei due non la dichiara. In entrambe le situazioni, come avviene nel linguaggio C, sarebbe buona pratica passare un secondo parametro contenente la size.

Proseguendo troviamo due utility che si occupano di interagire con la symbol table e di rilanciare opportunamente eventuali eccezioni. Dopodiché troviamo la funzione `check_global_assign_expr_type` che controlla ciò che è assegnabile, in fase di inizializzazione, a una variabile globale, ovvero solo costanti ed espressioni costanti.

Le funzioni `check_variable_type` e `check_parameter_type` si occupano di rilevare invece dichiarazioni di variabili o parametri di funzione prive di significato o non supportate, come una variabile avente tipo `void` o array multidimensionali. Non è supportata la dichiarazione di puntatori ad array, come ad esempio un `int (*p)[3]`.

Dopodiché inizia la parte dedicata al type checking vero e proprio, con funzioni adette al controllo mutuamente ricorsivo delle varie parti dell'AST. Questa sezione, sebbene sia abbastanza estesa, è piuttosto standard.

La funzione `check_declaration` esegue il type checking delle dichiarazioni di variabili locali e globali con annessa eventuale inizializzazione. Il linguaggio supporta l'inizializzazione di variabili scalari tramite assegnazione diretta di un valore e di array tramite lista di inizializzazione. Tramite apposita riscrittura del parser, il linguaggio è in grado di inferire la size di un array dalla lista di inizializzazione, se essa è non vuota. Una lista di inizializzazione vuota significa nessuna inizializzazione, è supportata ma in tal caso è obbligatorio specificare la size dell'array. È possibile utilizzare la lista di inizializzazione anche per variabili scalari, a patto che tale lista sia vuota, significante nessuna inizializzazione, oppure contenente un solo elemento.

La funzione `typecheck_statement`, per il type checking ricorsivo degli statement, utilizza due parametri aggiuntivi, `expected_ret_type` e `is_function_block`, per conoscere quale è il tipo dichiarato come tipo di ritorno della funzione contenente lo statement e per sapere se l'antenato diretto dello statement nell'AST era proprio la funzione. Il primo parametro è utile per il type checking dei valori restituiti, il secondo ha valenza solo nel caso di un `block` che normalmente è tenuto a generare un nuovo scope, ma non se è il diretto discendente di una funzione, la quale ha già provveduto alla creazione di un nuovo scope nel quale sono stati inseriti i parametri formali.

La funzione `typecheck_statement` è stata arricchita per poter rilevare anche la presenza di dead code. Sebbene la logica utilizzata sia abbastanza semplice, si è dimostrata efficace nel rilevare la presenza di dead code nei file di test `test-return1.mc`, `fail-dead1.mc` e `file-dead2.mc`. La funzione restituisce `true` nel caso in cui vi è la certezza che lo statement provochi la fuoriuscita dalla funzione nella quale è inserito, mentre se restituisce `false` lo statement potrebbe comunque "ritornare" oppure no. La proprietà da controllare in sé è ovviamente indecidibile, ma un possibile miglioramento è sicuramente quello di rilevare la presenza di condizioni costanti nei costrutti `if` e nel costrutto `while` (ricordiamo che `for` e `do-while` vengono trasformati in `while`).

Le possibili casistiche sono le seguenti:

- `if`, tale costrutto termina sicuramente la funzione se e solo se entrambe le diramazioni terminano
- `while`, questo costrutto non termina mai con certezza la funzione poiché anche se il suo corpo terminasse, la condizione potrebbe essere immediatamente falsa (questa è una marcata sotto-approssimazione migliorabile tramite l'analisi delle condizioni)
- `expression`, un expression-statement di per sé non può terminare la funzione
- `return`, un return statement certamente termina la funzione
- `block`, una sequenza di statement termina la funzione con certezza solo se almeno uno di essi termina con certezza

La rilevazione di dead code avviene proprio in quest'ultima situazione: se uno statement in un blocco termina con certezza e sono presenti altri statement dopo di esso, tali statement non potranno mai essere eseguiti.

È necessario spendere alcune parole anche sulla logica di type checking delle dichiarazioni di funzione. Durante il processo viene creato un nuovo scope nel quale vengono inseriti i parametri dopo gli opportuni controlli, il tipo della funzione viene generato tramite currying dei parametri come citato prima e la dichiarazione della funzione viene aggiunta allo scope globale. Il type checking del corpo della funzione viene invece posticipato, poiché in esso potrebbe essere presente l'invocazione di una o più funzioni dichiarate più avanti nel file `.mc`. Questa posticipazione consiste semplicemente nell'inserire il controllo del corpo

all'interno di una closure che sarà invocata in un secondo momento, dopo l'analisi delle dichiarazioni di tutte le funzioni presenti nel file.

Code generation

Nel file dedicato alla code generation troviamo tutte le funzioni, principali e secondarie, necessarie allo scopo. Troviamo immediatamente le definizioni dei tipi base secondo LLVM e una funzione di trasformazione dalle annotazioni di tipo nell'AST verso i tipi LLVM. Troviamo poi un contatore globale per assicurare nomi univoci da dare alle varie operazioni di costruzione delle istruzioni, contatore azzerato all'inizio della costruzione di ogni funzione. Le varie funzioni `build_*` sfruttano indirettamente tale contatore, e sono dei semplici wrapper attorno ai binding di ocaml alle primitive di LLVM.

In generale il codice generato fa un estensivo uso di operazioni da e verso la memoria principale: ogni volta che una variabile deve essere aggiornata viene immediatamente sovrascritta tramite una store, e ogni volta che una variabile deve essere letta verrà inserita una load corrispondente. Sebbene questo semplifichi concettualmente la generazione del codice, ad esempio non è mai necessario l'uso di una phi function, questo modo di procedere causa inevitabilmente un certo grado di inefficienza. Si lascia però la questione in mano alle varie fasi di ottimizzazione. Come per l'analisi semantica viene fatto uso di una symbol table, suddivisa in scope, nella quale immagazzinare l'indirizzo in memoria delle varie variabili e funzioni dichiarate nel programma, come anche dei parametri formali di queste ultime.

Incontriamo poi la funzione `get_value_at_addr`, la quale si occupa, dato l'indirizzo in memoria di una variabile, di ottenere il corrispondente valore costruendo una load. L'eccezione sono gli array, per i quali non ha senso caricare alcunché dalla memoria. Un array in quanto indirizzo del blocco contiguo di memoria è tutto ciò di cui c'è bisogno per i casi in cui il type system ne permette l'utilizzo.

Sono quindi presenti altre funzioni di ulitità, come quella per aggiungere una istruzione terminatrice ai blocchi che ne sono eventualmente sprovvisti, e una funzione per la valutazione di espressioni costanti.

Si passa quindi alle funzioni che generano il codice a partire dall'AST. Per quanto riguarda le espressioni, in particolare gli accessi alle variabili, le dereferenziazioni e gli accessi alle celle degli array, il risultato cambia se l'accesso derivava da una operazione di scrittura, ne qual caso richiediamo l'indirizzo, o in lettura, nel qual caso richiediamo il valore. Ecco che le funzioni generatrici del codice per le espressioni, `codegen_expression`, e per gli accessi, `codegen_access`, richiedono un parametro aggiuntivo, `should_ret_value`, che funge da discriminante tra le due casistiche.

L'accesso a una cella di un array è più delicato del previsto in quanto i parametri array di una funzione vengono fatti decadere, in stile C, a puntatori, perciò è necessario discriminare ulteriormente in tale situazione.

Per quanto riguarda la generazione del codice degli statement essa è piuttosto standard, articolata in determinate situazioni come la dichiarazione delle variabili locali o globali, ma senza particolari sorprese concettuali. Nella generazione del codice delle funzioni si utilizza lo stesso procedimento dell'analisi semantica: prima viene generato il minimo indispensabile per poter inserire nella symbol table ogni funzione dichiarata e poi si passa alla code generation dei corpi, la quale era stata rimandata sempre grazie ad una closure.

Language extension

Sono state implementate le seguenti estensioni:

- operatori di pre/post incremento/decremento, ovvero `++` and `--`, e le abbreviazioni degli assignment operator `+=`, `-=`, `*=`, `/=` e `%=`
- ciclo `do-while`, dichiarazione di variabili con inizializzazione e dichiarazioni multiple come `int i = 0, *j = &z;`
- rilevazione del dead code
- comma operator

Test

I test forniti sono stati suddivisi in due cartelle: `samples/fail` per i test che dovrebbero fallire, `samples/success` per i test che dovrebbero terminare con successo. Di questi ultimi, i test `test-ops2.mc` e `test-return1.mc` hanno subito delle modifiche evidenziate tramite commenti: il primo a causa di un uso errato dell'operatore di pre-decremento, il secondo a causa della presenza di dead code rilevato dall'analisi statica. Invece, i test presenti nella cartella `samples/mines` sono test aggiuntivi scritti per provare diverse funzionalità.

È possibile compilare ed eseguire un qualsiasi programma tramite lo script bash `compile.sh`:

```
bash compile.sh test/samples/path/to/file.mc [COMPILER_OPTIONS]
```

Lo script imposta un timeout all'esecuzione dei programmi per supportare i cicli infiniti. Sono infatti presenti altri due script bash, `launch_failing_tests.sh` e `launch_success_tests.sh`, che eseguono rispettivamente tutti i test case fallimentari e tutti i test case che dovrebbero compilare e girare con successo. In quest'ultimo caso in particolare è presente il file `test-ex7.mc` che contiene un ciclo infinito. Il test case `test-ex24.mc` invece è solo molto lento nel calcolo della funzione di Ackermann e il terzo ouput impiega un po' ad arrivare. Per passare da un test all'altro è necessario premere un qualsiasi carattere.

Documentation

La documentazione pare non essere generabile con il comando presente nel Makefile, poiché viene generata completamente vuota. Probabilmente il problema sta nel fatto che il package contenente l'intero progetto è privato, non pubblico. È stato quindi modificato il Makefile per supportare la generazione di documentazione per progetti privati, ma ho dovuto "hardcodare" un path specifico e non sono sicuro funzioni ovunque. Ad ogni modo non è stata seguita la convenzione dei commenti `odoc`, e i commenti risiedono completamente nei file `.ml`, perciò la documentazione generata non offre un gran valore aggiunto rispetto alla lettura del codice sorgente.