Lambda Semantics

Giuseppe Lomurno

1 INTRODUCTION

This project started with the objective of experimenting with all the phases of building a compiler and/or interpreter for a simple functional language.

The starting point was an existing project of mine that consisted of a correct-by-construction AST for single expressions and a small interpreter in Haskell. This core language has been extended with:

- A parser with some syntactic sugar (the grammar is available at Appendix A);
- A type checker that produces correct abstract syntax trees;
- Upgrades to the evaluation function and stepped evaluation;
- Compilation to LLVM intermediate representation, and in turn compilation to machine assembly via LLVM;
- JIT execution via LLVM;
- Some form of optimization directly on the language intermediate representations.

Although some difficulties were met dealing with immature Haskell extensions and with LLVM libraries for Haskell, all the major objectives have been achieved.

2 DESIGN

2.1 GENERAL DESIGN

The project language has been born as an implementation of the simply-typed lambda calculus with call-by-value semantic. This core has been extended with other primitive constructors such as pairs and a fixpoint operator for general recursion.

The abstract syntax tree of this functional language guarantees the well-formedness of expressions by construction in two ways:

- De Bruijn indexes to reference locally bound variables;
- Use of dependently typed functionalities, thanks to singletons library emulation. These ensure that the indices are in bounds and that the typing is correct with the typing environment as the type family index.

This tree is implemented by the `AST env ty` type family with `env :: [LType]` and `ty :: LType`, respectively the typing environment and the return type of the expression. Its constructor offer primitive support to integer and boolean literals and primitive operations on those via run-time support. There is also conditional evaluation with both return branches of the same type, lambda abstractions with a single argument, pairs and the fixpoint operator.

`AST` is not the only intermediate representation used by the project. For the frontend, I've built a non-typed syntax tree, the `Untyped` type, generated directly from the parser, with symbolic names, instead of indices, for variables. Using directly the typed AST for the parsing phases would have made the code difficult to read and to maintain. The need to convert directly to De Bruijn indices would have made parsing error related to variable particularly difficult to understand. Also, the incomplete support for dependent types would have required substantial modification to the parser types or a lot of boilerplate.

Another intermediate representation has been employed for the code generation backend, the `CodegenAST` type. This AST revealed itself particularly useful for the generation of the LLVM intermediate representation, because it does not directly support some constructs required for a functional language and, therefore, some transformations upon a less restrictive intermediate representation have made the compilation process easier to code. Some of the differences between the regular AST and the code generation AST are the support for top-level function declaration, let bindings computed without using lambda abstractions and recursive declaration without the need of a fixpoint operator.

2.2 LIMITATIONS

At the moment, no form of polymorphism is supported by the language syntax. Type inference is limited to the return type of expression, therefore it is mandatory to annotate each lambda abstraction with the type of its argument. Primitive operations are an exception, equality and disequality binary operators are polymorphic, nonetheless, the lack of ad-hoc polymorphism makes the supported type of these operators transparent to the users while writing the language code.

Primitive operations are indexed by the type of the arguments and the type of the return value. In this prototype stage, only binary and unary primitive operations are supported, however, support for primitives with arbitrary arity can be added with a process similar to the one used for the typing environment in the AST.

Although the `Fix` constructor of the `AST env ty` type supports whichever expression has the correct typing, during the type-checking phase this argument is restricted to lambda abstractions definition, the recursive function, effectively making this operator equivalent to a recursive let definition available in many functional languages, for example, the ML family. This choice is dictated by difficulties encountered during the compilation phase, detailed in section 3.3.

In the master branch of the repository, the language is restricted to pure expressions, therefore it is not possible to interact with the user or input devices. In the `io` branch, there is a tentative approach of impure expression definition with a monadic mechanism similar to the IO monad in Haskell, intending to encapsulate impure functions from pure ones.

3 IMPLEMENTATION

3.1 PARSING

The parsing phase is implemented with the megaparsec library, which offers a monadic interface to a `LL(k)` parser with the possibility of keeping some state during the parsing. The parser-combinators library offers prebuilt combinators for automatically generate correct parsers for operator precedence and left-recursive grammars. Nonetheless, the language grammar has been conceived trying to minimize backtracking, for example, by using distinct parenthesis for pair creation and precedence enforcement.

The grammar is based upon the common notation for the simply-typed lambda calculus, with some syntactic sugar such as the let binding, similar to the one available in Haskell. Let expressions are translated to lambda abstraction with application in this manner:

```
let x : t = e1 in e2 <=> (\x : t. e2) e1
```

And also the possibility of declaring lambda abstraction with multiple arguments:

```
\x1:t1,...,xn:tn. e <=> \x1:t1. ... \xn:tn. e
```

Valid identifiers must start with an ASCII letter followed by any number of alphanumeric characters or underscores. To avoid ambiguities, the grammar recognizes only positive integers and the minus symbol is translated to the primitive negation operation on integers, instead of parsing numbers such as `-42` as literals.

In the end, implementing the parser was extremily easy and declarative, thanks to the use of combinators offered by the adopted libraries. Also, parsing the input directly without an explicit tokenization phase allowed effortless meaningful error messages. The choice of a, potentially, less powerful parser (Alex-Happy offers Lex-Yacc functionalities for Haskell) was heavily influenced by this aspect, keeping in mind that backend operations, particularly optimization, can have worse performance than parsing.

3.2 INTERPRETER

The interpreter relies on a closed type family to convert from language types to native Haskell types. This guarantees that the values produced by the interpreter have correct typing even in Haskell. The evaluation function produces pairs of reduced ASTs and Haskell values.

The `eval` function evaluates completely an expression, as described in the big-step semantic available at Appendix C, using the call-by-value semantic for argument substitution. This function has nothing novel beside the conversion of lambda abstractions to Haskell function that takes another AST

as argument. The substitution function guarantees that the argument type is removed correctly from the typing environment and applies the correct shift to the De Bruijn indices, in a type-safe manner.

The `step` function makes a single step of beta-reduction, with the same semantics of `eval`. The fixpoint of this function is the same result that would have been obtained evaluating the expression with `eval`.

3.3 COMPILER

The compilation process starts with converting the AST IR to the code generation IR with top-level function declarations. All the AST constructors have a one-to-one correspondent with the addition of other constructors helpful to the code generation process.

The first transformation replaces series of single argument lambda abstractions to a single multi-argument lambda abstraction.

The second transformation removes the fixpoint operator and replaces calls to the recursive function with a token that, in turn, will be replaced during code generation with a call to the top-level function itself.

The third transformation is the closure conversion procedure. The LLVM IR does not support natively the concept of closure, fundamental for functional languages. One of the possible approaches was building a specific data structure that would hold a pointer to the environment of the closure and use some form of jump or trampolining for code execution. However, generating indirect jumps was not easy, in light of the lack of documentation or example in the LLVM Haskell bindings, therefore I've decided to opt for the closure conversion procedure. This algorithm adds all the values in the environment of a closure as explicit parameters of the function, thus making the closure a combinator. This procedure allows us to obtain function that perfectly corresponds to the function offered by the LLVM IR and potentially a greater propensity to the optimization supplied by the LLVM optimizer, at the cost of having an increased number of parameters and the lack of reuse in call chains. Anyhow, static analysis can remove unused parameters, both via the LLVM optimizer and via direct transformations on the lambda abstractions of the language.

The last transformation makes the obtained combinators into top-level declarations. This process is called lambda lifting.

After all these transformations the intermediate representation is ready for code generation. All the lifted lambdas are converted to regular LLVM functions, in reverse order of declaration, in order to avoid referencing function that still have not been generated. Finally, a single entry point function is generated, called `lab_main` keeping track of reference to the top-level declaration previously generated.

Integers and booleans are in direct correspondence to the type offered by the LLVM IR, while the unit type is generated as a null pointer, not used for any other case. The primitive operations correspond to the ones offered by LLVM, or a combination of them as in the case of boolean negation. Conditional evaluation uses the phi operator with two distinct blocks, keeping in mind that we cannot assume that the terminal block of a branch is the same

as the starting block. Pairs are reduced to simple structs with the fields of the corresponding type. Functions, as expected for a functional language, can be passed as argument to other functions. LLVM allows to pass pointers to function, therefore all the lifted combinators are passed as arguments in the form of pointers. Finally, the recursion tokens are replaced with references to the functions that declared them.

3.4 CSE

One of the fundamental goal of this project was experimenting with optimization transformations on the chosen intermediate representation. For this objective, I've chosen a data-flow optimization, the common subexpression elimination. The optimization has been implemented on the code generation ready abstract syntax tree, mainly because the complex Haskell extensions exploited for type-safety would have required new data structures for sets and maps on the `AST` IR. These are not in the scope of this projects and secondly, because of the chosen semantic, replacing common subexpressions with let would have required the addition of another constructor to the core that allowed for eager evaluation or call-by-name semantic. Anyhow the algorithm would have been the same for the `AST` IR, with the difference lying in the typing management.

The algorithm is composed of three distinct phases. First, the algorithm descends in the syntax tree, accumulating subexpression in hash sets and then inserting let expression for each reuse while ascending. Second, the subexpression lifted to a let expression are replaced with a reference to the generated lets. Third, multiple lets with the same subexpression could have been generated at various heights in the tree, thus only the top-most let is left.

This optimization is applied locally to each expression or declaration generated during the code generation process.

3.5 REPL

For language usability, a simple REPL has been made with the possibility of executing a series of command for each correctly analyzed expression. The basic functionalities consists of printing the final type of the expression, complete evaluation or stepped evaluation, generating the corresponding LLVM IR, executing the code via LLVM JIT compilation, or produce machine code also via LLVM.

At the moment, it is not possible to declare multiple expression, each visible to the others. For a complete documentation of the REPL, refer to Appendix D.

4 CONCLUSIONS

This project was not exempt from difficulties during the writing process. Primarily, the choice of using experimental extensions for type guarantees was harmful to coding times, in that much effort was expended to correctly codify collateral data structures more than focusing on code generation or optimizations.

Also, the use of LLVM was no easy matter, particularly regarding the JIT compilation and the marshalling of values, since the documentation is lacking and is difficult to translate the original LLVM documentation to actual Haskell code. On the other hand, the declarative monadic interface reduced the number of errors that affected the code generation and has been one of the strong points of this project. As important is the maturity of the Haskell ecosystem for building compilation and interpreters, particularly the vast number of parsing libraries, the REPL management and writing recursive algorithms for abstract syntax trees management.

Concerning the language design, adhere strictly to the syntax and semantics of the simply-typed lambda calculus allowed easy reasoning about the type system and the semantics of the language itself, but showed some deficiencies during the code generation process rather than during the evaluation process. Additional constructors revealed themself mandatory, nonetheless, the choice of producing multiple intermediate representation was a good choice to exploit the strengths of each representation and make assumptions on less type enforced representation, such as the code generation IR.

Positive was also the choice of a parsing library. The monadic parser allowed easy definitions and immediately useful error messages, thus gaining time for multiple grammar experimentations and prototypes. Though, I suggest reading the documentation of such parsers carefully and, when possible, use the combinators offered by these libraries to avoid infinite loops and costly backtrackings, common while managing operators with precedence.

A GRAMMAR

⟨lang⟩ ⊨ ⟨let⟩ | ⟨fix⟩ | ⟨lam⟩ | ⟨unop⟩ ⟨expr⟩ | ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
⟨expr⟩ ⊨ ⟨atom⟩ ⟨atom⟩ | ⟨atom⟩
⟨let⟩ ⊨ `let` ⟨arg⟩ `=` ⟨lang⟩ `in` ⟨lang⟩
⟨fix⟩ ⊨ `fix` ⟨lang⟩
⟨lam⟩ ⊨ `\` ⟨args⟩ `.` ⟨lang⟩
⟨atom⟩ ⊨ `()` | `(` ⟨lang⟩ `)` | ⟨int⟩ | ⟨bool⟩ | ⟨cond⟩ | ⟨ide⟩ | ⟨pair⟩
⟨args⟩ ⊨ ⟨arg⟩ | ⟨arg⟩ `,` ⟨args⟩
⟨arg⟩ ⊨ ⟨ide⟩ `:` ⟨types⟩
⟨types⟩ ⊨ ⟨type⟩ `->` ⟨type⟩ | ⟨type⟩
⟨type⟩ ⊨ `(` ⟨types⟩ `)` | `int` | `bool` | `unit` | `void` |
`{` ⟨types⟩ `,` ⟨types⟩ `}`
⟨cond⟩ ⊨ `if` ⟨lang⟩ `then` ⟨lang⟩ `else` ⟨lang⟩
⟨pair⟩ ⊨ `{` ⟨lang⟩ `,` ⟨lang⟩ `}`
⟨int⟩ ⊨ `0` | `1` | `...`
⟨bool⟩ ⊨ `true` | `false`
⟨ide⟩ ⊨ `[a-zA-Z][a-zA-Z0-9_]*`
⟨binop⟩ ⊨ `&` | `|` | `+` | `-` | `*` | `/` | `<` | `>` | `<=` | `>=` | `==` | `!=`
⟨unop⟩ ⊨ `-` | `~` | `fst` | `snd` |

B | TYPING RULES

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash \textbf{IntE}\ n : \textbf{LInt}}\ \textsc{TInt} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash \textbf{BoolE}\ b : \textbf{LBool}}\ \textsc{TBool} \qquad \frac{}{\Gamma \vdash \textbf{UnitE} : \textbf{LUnit}}\ \textsc{TUnit}$$

$$\frac{\text{unop} : \tau \to \sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \textbf{PrimUnaryOp}\ \text{unop}\ e : \sigma}\ \textsc{TUnOp} \qquad \frac{\text{binop} : \tau_1 \times \tau_2 \to \sigma \qquad \begin{matrix}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \tau_2\end{matrix}}{\Gamma \vdash \textbf{PrimBinaryOp}\ \text{binop}\ e_1\ e_2 : \sigma}\ \textsc{TBinOp}$$

binop is a supported binary operation

unop is a supported unary operation

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{Pair}\ e_1\ e_2 : \textbf{LProduct}\ \tau_1\ \tau_2}\ \textsc{TProd}$$

$$\frac{\Gamma \vdash e_1 : \textbf{LBool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{Cond}\ e_1\ e_2\ e_3 : \tau}\ \textsc{TCond}$$

$$\frac{\Gamma[x] = \tau}{\Gamma \vdash \textbf{Var}\ x : \tau}\ \textsc{TVar} \qquad \frac{\Gamma, \tau \vdash e : \sigma}{\Gamma \vdash \textbf{Lambda}\ e : \textbf{LArrow}\ \tau\ \sigma}\ \textsc{TLam}$$

$$\frac{\Gamma \vdash e_1 : \textbf{LArrow}\ \tau\ \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{App}\ e_1\ e_2 : \sigma}\ \textsc{TApp} \qquad \frac{\Gamma \vdash e : \textbf{LArrow}\ \tau\ \tau}{\Gamma \vdash \textbf{Fix}\ e : \tau}\ \textsc{TFix}$$

C | BIG-STEP SEMANTICS

$$\frac{}{\textbf{IntE}\ n \Downarrow \textbf{IntE}\ n}\ \textsc{SInt} \qquad \frac{}{\textbf{BoolE}\ b \Downarrow \textbf{BoolE}\ b}\ \textsc{SBool} \qquad \frac{}{\textbf{UnitE} \Downarrow \textbf{UnitE}}\ \textsc{SUnit}$$

$$\frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2'}{\textbf{Pair}\ e_1\ e_2 \Downarrow \textbf{Pair}\ e_1'\ e_2'}\ \textsc{SProd} \qquad \frac{e \Downarrow v \qquad \text{unop}(v) = v'}{\textbf{PrimUnaryOp}\ \text{unop}\ e \Downarrow v'}\ \textsc{SUnOp}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \text{binop}(v_1, v_2) \Downarrow v'}{\textbf{PrimBinaryOp}\ \text{binop}\ e_1\ e_2 \Downarrow v'}\ \textsc{SBinOp}$$

$$\frac{e_1 \Downarrow \textbf{BoolE True} \qquad e_2 \Downarrow v}{\textbf{Cond}\ e_1\ e_2\ e_3 \Downarrow v}\ \textsc{SCond1} \qquad \frac{e_1 \Downarrow \textbf{BoolE False} \qquad e_3 \Downarrow v}{\textbf{Cond}\ e_1\ e_2\ e_3 \Downarrow v}\ \textsc{SCond2}$$

$$\frac{}{\textbf{Lambda}\ e \Downarrow \textbf{Lambda}\ e}\ \textsc{SLam} \qquad \frac{e \Downarrow \textbf{Lambda}\ e' \qquad \text{subst}(\textbf{Fix}\ e, e') \Downarrow v'}{\textbf{Fix}\ e \Downarrow v'}\ \textsc{SFix}$$

$$\frac{e_1 \Downarrow \textbf{Lambda}\ e \qquad e_2 \Downarrow v_2 \qquad \text{subst}(v_2, e) \Downarrow v'}{\textbf{App}\ e_1\ e_2 \Downarrow v'}\ \textsc{SApp}$$

D | REPL

The REPL offers two mode: parse mode and command mode.

The loop starts in the parse mode, as can be seen by the `expr>` prompt. In this mode a single expression is expected and parsed, return eventual errors or the type of the expression:

```
expr> \x:int. x < 42
Expression parsed successfully with type :: (ℤ → 𝔹)
```

The command mode is available when an expression has been correctly parsed. When in command mode the prompt would be `cmd>`. The available commands are:

`untyped` For debug purposes, shows the untyped abstract syntax tree.

`typed` For debug pruposes, shows the typed abstract syntax tree.

`eval` Fully evaluates the expression, printing the resulting AST.

`step` Fully evaluates the expression, printint each beta-reduction applayed during the evaluation process.

`pretty` Pretty prints the expression (the output it is not meant to be parsed successfully).

`codegen` For debug purposes, pretty prints the code generation ready intermediate representation.

`llvm` Prints the LLVM IR result of the code generation phase.

`jit` Evaluates the expression with JIT compilation.

`compile` Produces assembly code via LLVM IR.

`expr` Returns the REPL to parse mode.

`quit` Quits from the REPL.