UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Bernardo Ferrari Mendonça

A programming language with refinement types and its LLVM-IR front end implementation.

Florianópolis
2024

Bernardo Ferrari Mendonça

A programming language with refinement types and its LLVM-IR front end implementation.

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Supervisor: Prof. Alcides Miguel Cachulo Aguiar Fonseca, Dr.
Co-supervisor: Prof. Rafael de Santiago, Dr.

Florianópolis
2024

Bernardo Ferrari Mendonça

A programming language with refinement types and its LLVM-IR front end implementation.

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 8 de Julho de 2024.

Prof^{a}. Lúcia Helena Martins Pacheco, Dr.
Coordenadora do Curso

Banca Examinadora:

Prof. Alcides Miguel Cachulo Aguiar Fonseca, Dr.
Orientador
Universidade de Lisboa

Prof. Rafael de Santiago, Dr.
Coorientador
Universidade Federal de Santa Catarina

Prof^{a}. Jerusa Marchi, Dr.
Avaliadora
Universidade Federal de Santa Catarina

This work is dedicated to my mother, Marcela Ferrari, my sister Camila Ferrari, and to my grandparents Márcio Ferrari and Cláudia Ferrari who plowed the field where I bloom.

RESUMO

Esta tese apresenta o design e a implementação do Ekitai, uma linguagem de programação que integra tipos refinados com um front end LLVM-IR. O objetivo principal é aproveitar os tipos refinados para melhorar a segurança de tipos e a otimização durante a geração de código. Exploramos a teoria e os aspectos práticos da incorporação de tipos refinados, que permitem expressar invariantes mais precisas nos tipos. A integração com LLVM-IR demonstra como esses tipos podem ser usados para guiar os processos de otimização e verificação no pipeline de compilação. A avaliação destaca os benefícios e desafios dessa abordagem, fornecendo insights para melhorias e extensões futuras.

Palavras-chave: Linguagem de Programação. Tipos Refinados. Representação Intermediária de Código. Compilador. LLVM. LLVM-IR.

ABSTRACT

This thesis presents the design and implementation of Ekitai, a programming language that integrates refinement types with a LLVM-IR front end. The primary objective is to leverage refinement types to enhance type safety and optimization during code generation. We explore the theory and practical aspects of incorporating refinement types, which allow for expressing more precise invariants in types. The integration with LLVM-IR demonstrates how these types can be used to guide optimization and verification processes in the compilation pipeline. The evaluation showcases the benefits and challenges of this approach, providing insights into future improvements and extensions.

Keywords: Programming Language. Refinement Types. Intermidiate Code Representation. Compiler. LLVM. LLVM-IR.

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

As stated by Aho et al. (2006), “Programming languages are notations to describe computations to people and to machines”. These notations can take numerous forms. They range from lower-level languages, such as machine code ready to be executed by a specific machine, to a higher-level language, such as C, Java, Rust, Haskell and Ml. Lower-level languages like machine code are very verbose in how they describe computations; usually describing directly to the machine when and how to execute each computation through simple notations like: add the values from two data locations, compare two values, jump the next 4 instructions, and so on (AHO et al., 2006). Whereas, in a higher-level language, we can describe computations in a more abstract set of notations, such as functions and types, without needing to expose details from the specific machine that will execute them (AHO et al., 2006). Hence, using higher-level programming languages eases how people can describe computations to machines and to one another (AHO et al., 2006). However, in order to translate a higher-lever programming language to machine code capable of running on a machine’s processor, we need to design and build programs called compilers (AHO et al., 2006).

A compiler is a program that receives as input a program written in a *source* language and translates it to a semantically equivalent program written in a *target* language (AHO et al., 2006). This is what enables us to write programs in higher-level languages that are able to execute at a machine’s processor. A program written with a higher-level language such as C is fed into a compiler like Clang, then Clang translates the provided source program to a program in a target language, like Intel’s x86 processor’s machine code, ready to be executed. During the translation process between the *source* and *target* languages, the compiler goes through two major execution steps: the analysis step, and the synthesis step (AHO et al., 2006). The analysis step, called the compiler’s *front end*, organizes the information included in the source program into a grammatical structure, and then uses this grammatical structure, together with some metadata collected during its construction, to build what is called an intermediate representation (AHO et al., 2006). Furthermore, the synthesis step, called the compiler’s *back end*, uses this intermediate representation to compute the desired target program (AHO et al., 2006).

Though it is possible to build a compiler that translates directly to a target machine code, this hinders portability and modularity (APPEL; PALSBERG, 2003). Suppose we wish to implement a compiler for the source language i to the target machine language j, we can implement just the compiler’s *front end* for i and use a proven working *back end* for j (AHO et al., 2006). Therefore, if we wish to implement compilers for n different programming languages to m different machine languages, we can avoid building $n \times m$ compilers building n *front ends* and m *back ends* (AHO et al., 2006).

If we give the analysis-synthesis model of a compiler a more fine-grained look, we can identify that the compiler's front and back end operate as a series of phases, each one transforming one intermediate representation to another in order to further advance the computation of the target program (AHO et al., 2006). The analysis step, or front end, may be subdivided into: a lexical analyzer, a syntax analyzer, a semantic analyzer and an intermediate code generator; also, the synthesis step may be subdivided into: machine-independent code optimization, code generation, and machine-dependent code optimization (AHO et al., 2006). The different phases and the intermediate representations between them can be seen at Figure 1.

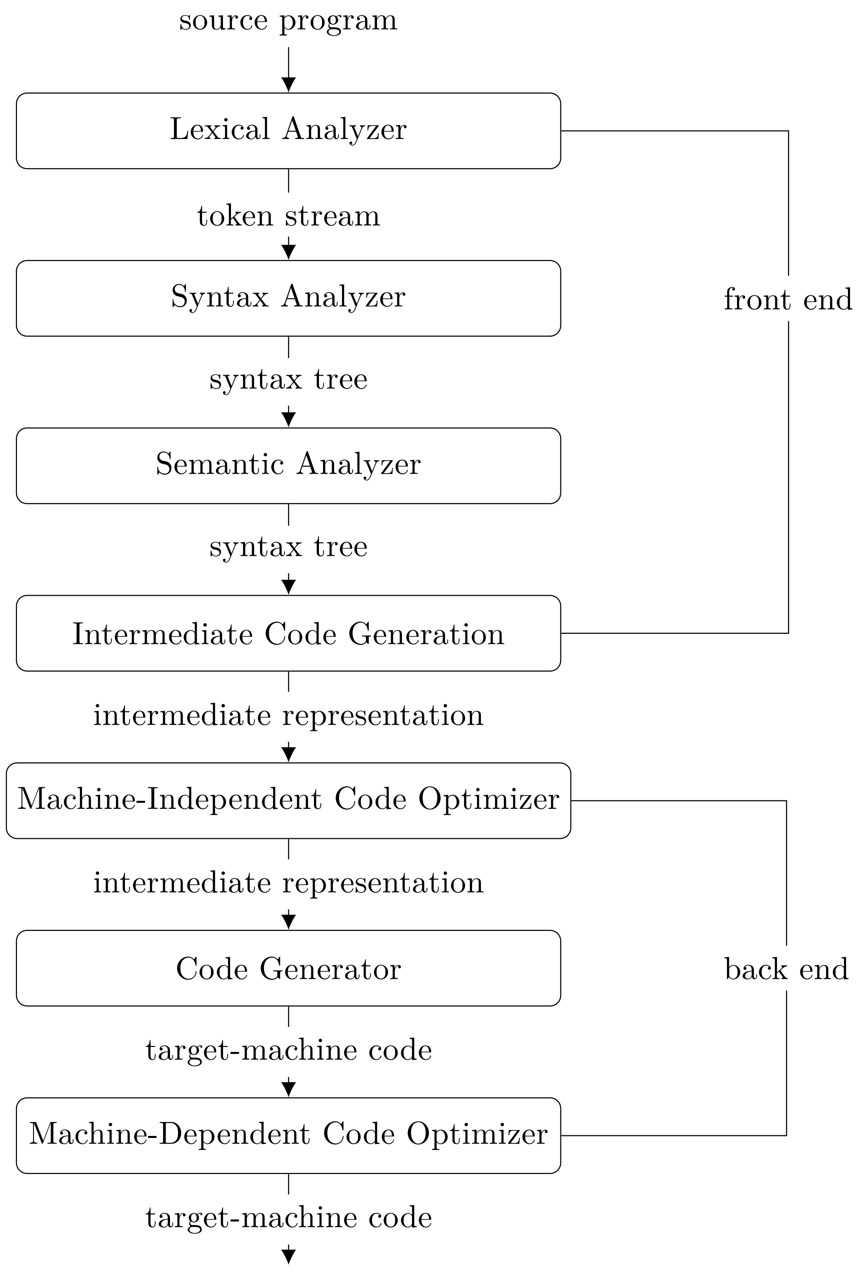


Figure 1 – Phases of a compiler and the intermediate representations between them. Adapted from Aho et al. (2006).

According to Appel and Palsberg (2003), “An intermediate representation (IR) is a kind of abstract machine language that can express the target-machine operations without committing to too much machine-specific detail”. The authors continue by adding that the IR “is also independent of the details of the source language” (APPEL; PALSBERG, 2003). This means that the abstract notations exclusive to a higher-level language are handled by the front end of a compiler so that, by the time the source program is transformed into the IR, the computations described by the IR are semantic equivalent to the computations described by the source program but are now in the notations of an abstract machine language. Although the semantics of the computations are the same, there may be semantics in the higher-level language that are not present in the IR. The front end of a compiler is then responsible to check if all semantic aspects of the source program are sound to the source language specifications before the generation of the IR (AHO et al., 2006). The authors explain that the checks made during compilation are called *Static Checks*, and they are not only capable of assuring that the source program can be successfully compiled, but have the potential to catch programming errors early, before the program can be executed (AHO et al., 2006). One of the static checks executed during compilation is *type checking* and is part of the semantic analysis phase of the compiler’s front end (APPEL; PALSBERG, 2003).

The type checking executed during the semantic analysis phase is designed in accordance with the source language’s type system. Pierce (2002) defines a type system as: “A tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute”. Hence, a phrase written with higher-level notations such as

$$\texttt{x * 30} \tag{1.1}$$

may be classified to compute a value of type `Int`, written

$$\texttt{x * 30 : Int}$$

meaning that (1.1) is a phrase that computes a mathematical integer value. As a counter example, if defined by the type system that the multiplication between a value of type `Bool` and a value of type `Int` was not allowed, and we had both classifications

$$\texttt{x : Bool} \qquad \text{and} \qquad \texttt{30 : Int}$$

the phrase in (1.1) would be semantically unsound and should be indicated as a type error by the type checker.

According to Jhala and Vazou (2020), type systems are mainly used to describe valid sets of values that can be used for different computations so that the compiler can eliminate a variety of possible run-time errors before the target program execution. Type systems like the ones used by popular modern programming languages, such as C#, Haskell, Java, OCaml, Rust and Scala, have similar kinds of rules and are the most widespread tool used to guarantee the correct behavior of a program (JHALA; VAZOU, 2020). Jhala and Vazou (2020) affirm that, although type systems are widespread and effective, well-typed programs do go wrong. The authors elaborate a few wrong behaviors that are common to the most popular type systems. Within them, we have:

- **Division by zero:** Constraining the types of the division operation to `Int` does not protect the program to execute a division by zero at run-time, and it does not guarantee that the arithmetic operations will not under- or over-flow (JHALA; VAZOU, 2020).
- **Buffer overflow:** Constraining the index of the access of an `Array` or `String` to `Int` does not protect the program to try to access data from beyond the data structure's end (JHALA; VAZOU, 2020).

An effort can be made while designing a type system so that they can further restrict the values of certain types. We can extend a type system to further *refine* its types with logic predicates and this method is called *Refinement types with predicates* (JHALA; VAZOU, 2020). It allows programmers to constrain existing types by using predicates to assert desired properties of the values they want to describe (JHALA; VAZOU, 2020). For example, while `Int` types can assume any integer values, we can write the refined type

```
type Nat = {v:Int | 0 <= v}
```

where the newly defined type `Nat` will only be able to assume positive integer values. Alone, this refinements may seam just a gimmick, but combined with functions the programmer can describe precise contracts that describe the functions legal inputs and outputs (JHALA; VAZOU, 2020). For example, the author of an `array` library may specify the functions signatures types

```
fn size: x:array(a) -> {v:Nat | v = length(x)}
fn get: x:array(a) -> {v:Nat | v < length(x)} -> a
```

where `size` and `get` are functions and `x` is the name of the first argument. In this type system, a call to `size(x)` returns a value s of type `Nat` constrained to a single value equal to the length of `x`; hence, the type of s constrains s to the exact length of `x`. Furthermore, a call to `get(x, i)` requires the index `i` to be within the bounds of `x`. Given these definitions, the refinement type checker can then prove, during the analysis phase (i.e., at compile-time), that the contracts of both `size` and `get` will not be violated, ensuring all array access to be sated when executing the target program (i.e., at run-time) (JHALA; VAZOU, 2020).

1.1 MOTIVATION AND RESEARCH PROBLEM

There is an increasing number of uses of refinement types being implemented on top of existing languages. For example: the work of Vazou, Seidel, and Jhala (2014) presenting LiquidHaskell as refinement types for the Haskell language; the work of Vekris, Cosman, and Jhala (2016) integrating refinement types for the TypeScript language; the work of Sammler et al. (2021) integrating refinement types for the C language; and, the work of Kazerounian et al. (2018) integrating refinement types for the Ruby language.

Although refinement types have been proved useful in improving the static checking capabilities of higher level languages, there is a lack of research on bringing refinement types to the intermediate code generation phase of a compiler's front end. Hence, we state the research problem in the form of the question: What can we discover when we add refinement types to the intermediate code generation phase of a compiler's front end?

1.2 GOALS

The primary goal of this work is to discuss and validate the following thesis: A front-end for a higher-level language with refinement types can make use of refinement types to allow optimizations opportunities during intermediate code generation not present in higher-level languages without refinement types.

For allowing this discussion, we will be designing a language with refinement types called *Ekitai* and implementing its front end. Designing a new language is particularly advantageous because we have full control of all the language features being implemented and allow us to incrementally design the language, the refinement type system, and the intermediate code generation, one feature at a time.

1.2.1 Specific Goals

The specific research artifacts constructed by this work that allow the discussion of the thesis are:

- The specification of *Ekitai's* lexical elements;
- The specification of *Ekitai's* syntax;
- The specification of *Ekitai's* type system with refinement types;
- The implementation of *Ekitai's* front end including:
    - A lexical analyzer;

  - A syntactic analyzer;
  - A semantic analyzer with a type checker;
  - An intermediate code generator;
- The implementation of optimizations during intermediate code generation.

1.3 METHODOLOGY

The aim of this work is to find optimizations opportunities during intermediate code generation of a higher-level language with refinement types. In order to achieve the specific goals specified in Section 1.2.1 we will employ different methods during research.

In the specification of the *Ekitai* programming language aspects we will employ techniques from the established literature, such as books and articles, on language design and compiler construction. Also, in order to specify the type system with refinement types we analyze the recent publications about refinement types in the ACM Sigplan's conferences and journals. Whereas in the implementation of *Ekitai's* front end we will develop a front end using the Rust programming language employing techniques from the established literature, such as books and articles, and from open-source implementations of modern industry programming language compilers and tools.

1.4 STRUCTURE OF THE WORK

In Chapter 2 we will introduce the background knowledge needed to design a higher level language's front end. Furthermore, in Chapter 3, we explore how to add refinements to a lambda calculus language. In Chapter 4, we explore how to use the LLVM intermediate representation. In Chapter 5, we describe the proposed *Ekitai* language, its frontend implementation, and major results of the research. Than, in Chapter 6, we talk about future work ideas and bring closure to the thesis.

2 A REVIEW ON FRONT END DESIGN AND IMPLEMENTATION

In this chapter we present a brief review of the background needed to build a compiler's front end. We begin by presenting the formal definition of a language in Section 2.1. Then, give a brief overview of what is a lexical analyzer and how it is constructed in Section 2.2. We continue by presenting the aspects of a syntax analyzer in Section 2.3. Furthermore, we go on to explore the semantic analyzer and the tools used to formalize the type system in Section 2.4. And then, explore the aspects of intermediate code generation in Section 2.5.

2.1 STRINGS AND LANGUAGES

As stated by Sipser (2012), "strings of characters are fundamental building blocks in computer science". In order to define what a string of characters is, Sipser (2012) defines an *alphabet* to be any nonempty finite set composed by its *symbols*. The author elaborates that a *string over an alphabet* is a finite sequence of symbols from that alphabet (SIPSER, 2012). For example, if we build the English alphabet as the set of symbols

$$\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \ldots, \mathsf{x}, \mathsf{y}, \mathsf{z}\}$$

then, the word `compiler` would be a string over Σ.

Formally, a string s over an alphabet Σ is a string s such that $s \in \Sigma^*$, called the Kleen closure of Σ (HOPCROFT; MOTWANI; ULLMAN, 2007). In order to perform the inductive construction of Σ^* we need to first define the empty string, written ε, then, define the concatenation operation $uv = s$, where

$$u, v, s, w \in \Sigma^*$$

$$u = u_1 u_2 u_3 \ldots u_i \quad \text{where} \quad u_1 u_2 u_3 \ldots u_i \in \Sigma \quad \text{and} \quad i \geq 0$$

$$v = v_1 v_2 v_3 \ldots v_i \quad \text{where} \quad v_1 v_2 v_3 \ldots v_i \in \Sigma \quad \text{and} \quad j \geq 0$$

$$s = u_1 u_2 u_3 \ldots u_i v_1 v_2 v_3 \ldots v_i$$

$$u\varepsilon = u \quad \varepsilon u = u$$

$$(uv)w = u(vw)$$

Next, we perform the inductive construction

$$\begin{aligned} \Sigma^0 &= \{\varepsilon\} \\ \Sigma^1 &= \Sigma \\ \Sigma^{i+1} &= \{uv \mid u \in \Sigma^i \text{ and } v \in \Sigma\} \\ \Sigma^* &= \bigcup_{i \geq 0} \Sigma^i \end{aligned}$$

This construction shows that Σ^* is composed of all strings of finite size for all permutations of the alphabet (HOPCROFT; MOTWANI; ULLMAN, 2007).

After the definition of the set of all strings over an alphabet, we can broadly define a language L as a proper subset of Σ^*, written $L \subseteq \Sigma^*$, and, define the operations on languages as such: given two languages L and K we have that

$$
\begin{aligned}
L \cup K &= \{s \mid s \in L \text{ or } s \in K\} \\
LK &= \{uv \mid u \in L \text{ and } v \in K\} \\
L^* &= \bigcup_{i \geq 0} L^i \\
L^+ &= \bigcup_{i \geq 1} L^i
\end{aligned}
$$

Although having a simple definition, proving that a string s belongs to a language L turns out to be rather challenging.

We end up using different languages with different symbols for lexical, syntax and semantic analysis in order to verify the source program. Although, for example, the lexer may take as its symbols the characters present in the encoding of its input file (e.g., UTF-8 and ASCII), the parser may take the token names outputted by the lexer as its symbols. In the next Sections 2.2 to 2.4, we explore the different formal tools the different analysis steps use to prove that a string of their respective symbols is a valid source program and how they produce an output for the next step in compilation.

2.2 LEXICAL ANALYSIS

The first phase of a compiler is called lexical analysis. The lexical analyzer will read a stream of characters from the source program and group the characters into sequences called *lexemes* according to the *patterns* defined for each *token name* (AHO et al., 2006). Aho et al. (2006) describes that, for each lexeme grouped by reading the input character stream, the lexical analyzer will generate a *token* which constitutes a pair of the token name and some optional metadata (e.g., the start and end position of the respective lexeme in the input character stream) written

$$\langle \textit{token name},\ \textit{metadata} \rangle$$

For example, upon analyzing the following program:

```
let a = b + 60 / c
```

A lexical analyzer may output the following token stream:

$$\begin{array}{lllll}
\langle\texttt{let}, & \{\mathit{lexeme}\text{: }\texttt{let}, & \mathit{begin}\text{: }0, & \mathit{end}\text{: }3\}\rangle \\
\langle\texttt{identifier}, & \{\mathit{lexeme}\text{: }\texttt{a}, & \mathit{begin}\text{: }4, & \mathit{end}\text{: }5\}\rangle \\
\langle\texttt{=}, & \{\mathit{lexeme}\text{: }\texttt{=}, & \mathit{begin}\text{: }6, & \mathit{end}\text{: }7\}\rangle \\
\langle\texttt{identifier}, & \{\mathit{lexeme}\text{: }\texttt{b}, & \mathit{begin}\text{: }8, & \mathit{end}\text{: }9\}\rangle \\
\langle\texttt{+}, & \{\mathit{lexeme}\text{: }\texttt{+}, & \mathit{begin}\text{: }10, & \mathit{end}\text{: }11\}\rangle \\
\langle\texttt{number}, & \{\mathit{lexeme}\text{: }\texttt{60}, & \mathit{begin}\text{: }12, & \mathit{end}\text{: }14\}\rangle \\
\langle\texttt{/}, & \{\mathit{lexeme}\text{: }\texttt{/}, & \mathit{begin}\text{: }15, & \mathit{end}\text{: }16\}\rangle \\
\langle\texttt{identifier}, & \{\mathit{lexeme}\text{: }\texttt{c}, & \mathit{begin}\text{: }17, & \mathit{end}\text{: }18\}\rangle
\end{array} \tag{2.1}$$

In this case, the metadata collected by the analyzer is very pedantic including even redundant data as the lexemes for simple patterns of the token names `=`, `+` and `/`. When no metadata is needed it can be omitted from the token notation (e.g., $\langle\texttt{\{}\rangle$).

In order to classify a lexeme to be of a given token name we employ the use of *patterns*, and one of the important notations for the description of token patterns are regular expressions (AHO et al., 2006). Regular expressions are very effective in specifying the type of patterns usually needed to classify lexemes into tokens and can be used for automatic generation of a lexical analyzer (AHO et al., 2006). A simple description of the patterns for the tokens used in the example above can be given by regular expressions with rules of the form *pattern name* $\rightarrow$ *regular expression* as follows:

$$\begin{aligned}
\texttt{id} &\rightarrow [\texttt{A-Za-z}]^{+} \\
\texttt{number} &\rightarrow [\texttt{0-9}]^{+} \\
\texttt{let} &\rightarrow \texttt{let} \\
\texttt{=} &\rightarrow \texttt{=} \\
\texttt{/} &\rightarrow \texttt{/} \\
\texttt{+} &\rightarrow \texttt{+}
\end{aligned} \tag{2.2}$$

Regular expressions are built recursively out of smaller regular expressions, and each regular expression r describes a language L, written $L(r)$, which is also defined recursively from r's sub-expressions (AHO et al., 2006). We can define regular expressions starting with the regular expression ε, for describing the language $L(\varepsilon) = \{\varepsilon\}$, then, defining a regular expression a for all $s \in \Sigma$ where $L(a) = \{s\}$ (AHO et al., 2006). Then, building the induction supposing that r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively, as

$$\begin{aligned} r \mid s &= L(r) \cup L(s) \\ rs &= L(r)L(s) \\ r^{*} &= L(r)^{*} \\ r^{+} &= L(r)^{+} \end{aligned}$$

Using these operations we can build all regular expressions (AHO et al., 2006). Furthermore, a group of unions $s_1 \mid s_2 \mid \cdots \mid s_k$ can be abbreviated as $[s_1 s_2 \ldots s_k]$ (AHO et al., 2006).

The notation presented in (2.2) is defined by Aho et al. (2006) as *regular definitions* where, if Σ is an alphabet, then a regular definition is a sequence of definitions of the form:

$$\begin{aligned} d_1 &\rightarrow\ r_1 \\ d_2 &\rightarrow\ r_2 \\ &\cdots \\ d_n &\rightarrow\ r_n \end{aligned}$$

where, each definition name d_i is a new symbol not present in Σ and unique from the other definition names. Also, each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$ such that r_i can use the definitions above him as regular expressions (AHO et al., 2006). In order to facilitate the use of definition names in a regular expression, modern tools often write the definition name between the markers `[:` and `:]` as in

$$\begin{aligned} \texttt{digit} &\rightarrow\ [\texttt{0-9}] \\ \texttt{letter} &\rightarrow\ [\texttt{a-zA-Z}] \\ \texttt{id} &\rightarrow\ \texttt{[:letter:]}[\texttt{[:letter:][:digit:]}]^{*} \\ \texttt{number} &\rightarrow\ \texttt{[:digit:]}^{+} \\ \texttt{let} &\rightarrow\ \texttt{let} \\ \texttt{=} &\rightarrow\ \texttt{=} \\ \texttt{/} &\rightarrow\ \texttt{/} \\ \texttt{+} &\rightarrow\ \texttt{+} \end{aligned} \tag{2.3}$$

Formally, the lexer for a programming language is trying to prove that a given input string s belongs to a language L. For example, a programming language could define the regular definitions `id`, `number`, `let`, `=`, `/`, and `+`, from (2.3), as the patterns for its token names (`digit` and `letter` serve only as building blocks). Then, the language L for the parser of such programming language is constructed by making the union of the languages described by all the token name's regular expressions and applying the Kleen closure to generate the set of all permutations getting

$$L = (L(\texttt{id}) \cup L(\texttt{number}) \cup L(\texttt{let}) \cup L(\texttt{=}) \cup L(\texttt{/}) \cup L(\texttt{+}))^{*}$$

During the lexer execution, every time the lexer identifies a lexeme from the input string for a given token name's regular expression, the lexer outputs a token for that lexeme and consumes the lexeme from the input string (AHO et al., 2006). If an input string does match a lexeme for any token name regular expression, the lexer should report the error to the compiler's user (AHO et al., 2006).

The regular definition rules of token names can then, either be used by lexical analyzer generators to automatically generate a program for lexical analysis, or, be used as the formal specification for handwritten lexers to be based upon (AHO et al., 2006). In both cases the output will either be the list of tokens or the lexical error encountered during the reading of the input string.

2.3 SYNTAX ANALYSIS

The second phase of a compiler is called syntax analysis, or *parsing*. The *parser* receives as input a token stream produced by the lexical analyzer and creates a tree-like intermediate representation that is constrained by a particular grammatical structure (AHO et al., 2006).

If we give the token stream (2.1) as input to a parser it may produce the *parse tree* structure found on Figure 2. This structure has more information than the linear token stream it received as input. For example, it is prepared in such a way to preserve the order of operations from classic arithmetic. Consequently, the tree is composed of two interior nodes labeled `Expr` for binary operations: the bottom one, denoting the sub-expression

$$\texttt{Expr}_{bottom} = \texttt{60 / c}$$

and the top one, denoting the whole expression

$$\texttt{Expr}_{top} = \texttt{a + Expr}_{bottom} = \texttt{a + 60 / c}$$

This structure makes explicit that we must first evaluate the result of $\texttt{Expr}_{bottom}$, dividing `60` by `c`, before we can evaluate the result of $\texttt{Expr}_{top}$, adding `a` to the result of $\texttt{Expr}_{bottom}$.

2.3.1 Precise definition of context-free grammars and ambiguity

Every programming language has precise rules that prescribe the correct syntactic structure of its programs (AHO et al., 2006). In order to formally describe the rules of such syntactic structure, we can use a context-free grammar (AHO et al., 2006). A context-free grammar is a finite set of constructs that allows us to build the set of all strings, usually called *sentences*, of a given context-free Language (SIPSER, 2012). They have four components:

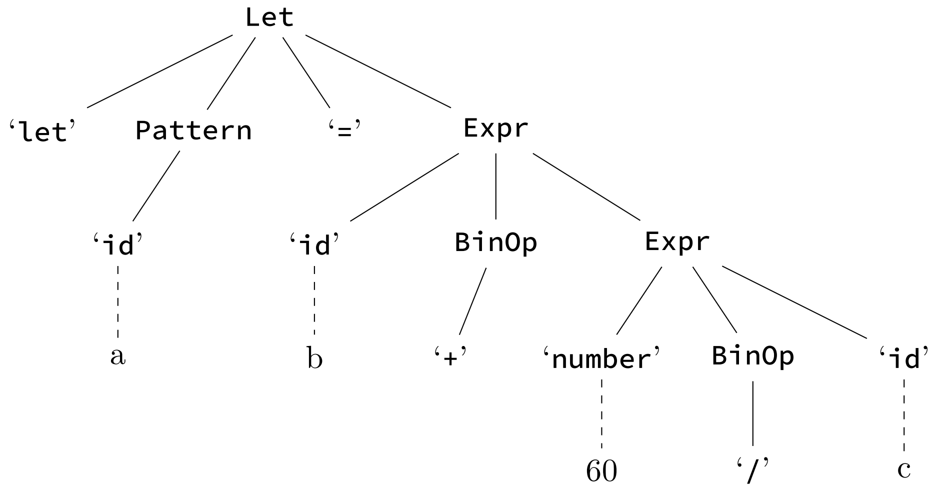


Figure 2 – A possible parse tree output for the token stream (2.1). Leafs are shown as the token names from the token stream. Here lexemes for non-trivial tokens are appended with dashed lines underneath token names for example purposes.

- A finite set Σ of *terminal* symbols, usually called tokens, composed by the set of token names defined in the lexical aspects of the language (AHO et al., 2006);
- A finite set V of *nonterminals*, or *variables*, disjoint from the set of terminals, usually called *syntactic variables* (AHO et al., 2006);
- A finite set R of *productions*, or *rules*, where each production is of the form

  $$\texttt{Variable} ::= \texttt{s}$$

  where `V` is the variable at the production's *head*, and $s \in (\Sigma \cup V)^*$ is the string at the production's *body* (SIPSER, 2012);
- And, a designation of a nonterminal S as the *start* of the grammar (AHO et al., 2006).

A simple expression language could have its syntactical structure formalized by the following grammar

$$\begin{aligned} \texttt{Expr} &::= \texttt{Expr}\ \texttt{BinOp}\ \texttt{Expr} \\ &\mid \text{‘}\texttt{(}\text{’}\ \texttt{Expr}\ \text{‘}\texttt{)}\text{’} \\ &\mid \text{‘}\texttt{number}\text{’} \\ &\mid \text{‘}\texttt{id}\text{’} \\ \texttt{BinOp} &::= \text{‘}\texttt{+}\text{’} \mid \text{‘}\texttt{-}\text{’} \mid \text{‘}\texttt{*}\text{’} \mid \text{‘}\texttt{/}\text{’} \end{aligned} \tag{2.4}$$

where: terminals have the token names between quotes (e.g., ‘`id`’); nonterminals have the syntactic variables as normal words (e.g., `Expr`); and, the designation of the starting variable is done by the variable at the head of the first production (e.g., in the example above the starting nonterminal is `Expr`). Furthermore, multiple productions can be abbreviated by using the *or* operator denoted by '|' turning

$$\begin{aligned}\texttt{V} &::= s_1\\ \texttt{V} &::= s_2\end{aligned}$$

into

$$\texttt{V} ::= s_1 \mid s_2$$

In order to show that a particular string of terminal symbols is in the language formalized by a context-free grammar, we can perform a derivation (SIPSER, 2012). A derivation starts with, first, writing the start variable (SIPSER, 2012). Then, second, we choose any variable from the string written so far and a production with the same variable as head replacing the chosen variable with the body of the chosen production (SIPSER, 2012). Then, we repeat the second step until there is no more variables and the string is formed only by terminals (SIPSER, 2012).

Formally, Sipser (2012) defines that u *derives* v, written $u \Rightarrow^* v$ where $u, v \in (\Sigma \cup V)^*$, if $u = v$ or if there exists strings $s_1, s_2, s_3, \ldots, s_k \in (\Sigma \cup V)^*$ for $k \geq 0$ where

$$u \Rightarrow s_1 \Rightarrow s_2, \Rightarrow s_3, \Rightarrow \cdots \Rightarrow s_k \Rightarrow v$$

We can then define that the language L of the grammar is the set $L = \{s \in \Sigma^* \mid S \Rightarrow^* s\}$, meaning L is the set of all strings s composed only by terminal symbols that can be derived from the starting variable S.

Take, for example, the string $s =$ `id + number * id` and the language L specified by grammar (2.4). We can proof that $s \in L$ by showing that `Expr` $\Rightarrow^* s$, so we start by writing the start variable

$$\texttt{Expr}$$

then choose a production with `Expr` as head and substitute `Exrp` with the production's body. We know there are two binary operations, so we choose the production which its body is `Expr BinOP Expr` producing the derivation

$$\texttt{Expr} \Rightarrow \texttt{Expr BinOp Expr}$$

For the next derivation step, we are open to choose what variable to derivate first. Although we could choose any variable from the string during derivation, we will stick to leftmost derivations, which mandates that we choose the leftmost variable, highlighted in **bold**, of the string to perform the derivation. So, we could have the leftmost derivation

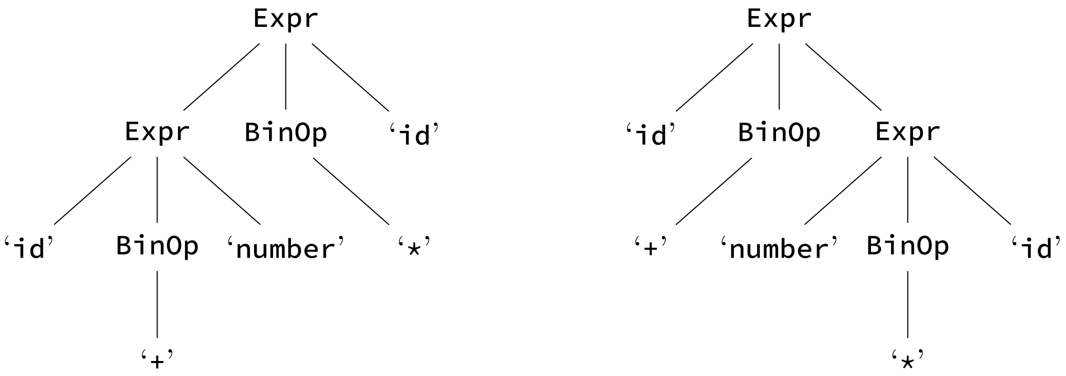
$$\begin{aligned}\textbf{\texttt{Expr}} &\Rightarrow \textbf{\texttt{Expr}}\ \texttt{BinOp Expr}\\ &\Rightarrow \textbf{\texttt{Expr}}\ \texttt{BinOp Expr BinOp Expr}\\ &\Rightarrow \texttt{`id'}\ \textbf{\texttt{BinOp}}\ \texttt{Expr BinOp Expr}\\ &\Rightarrow \texttt{`id' `+'}\ \textbf{\texttt{Expr}}\ \texttt{BinOp Expr}\\ &\Rightarrow \texttt{`id' `+' `number'}\ \textbf{\texttt{BinOp}}\ \texttt{Expr}\\ &\Rightarrow \texttt{`id' `+' `number' `*'}\ \textbf{\texttt{Expr}}\\ &\Rightarrow \texttt{`id' `+' `number' `*' `id'}\end{aligned}\tag{2.5}$$

thus, proving that the $s \in L$ but, we could also have a second leftmost derivation

$$\begin{aligned}\textbf{\texttt{Expr}} &\Rightarrow \textbf{\texttt{Expr}}\ \texttt{BinOp Expr}\\ &\Rightarrow \texttt{`id'}\ \textbf{\texttt{BinOp}}\ \texttt{Expr}\\ &\Rightarrow \texttt{`id' `+'}\ \textbf{\texttt{Expr}}\\ &\Rightarrow \texttt{`id' `+'}\ \textbf{\texttt{Expr}}\ \texttt{BinOp Expr}\\ &\Rightarrow \texttt{`id' `+' `number'}\ \textbf{\texttt{BinOp}}\ \texttt{Expr}\\ &\Rightarrow \texttt{`id' `+' `number' `*'}\ \textbf{\texttt{Expr}}\\ &\Rightarrow \texttt{`id' `+' `number' `*' `id'}\end{aligned}\tag{2.6}$$

thus, having two leftmost derivations that shows the string belongs to the grammar's described language.

A grammar is said to be *ambiguous* if it has a non-unique leftmost or rightmost derivation for any string of the language it describes (AHO et al., 2006). Therefore, grammar (2.4) is clearly ambiguous given derivations (2.5) and (2.6). From the parse trees in Figure 3 we can visualize that the order of operations differ between the derivations of `id + number * id`. In Figure 3a we have `(id + number) * id` and in Figure 3b we have `id + (number * id)` which have different mathematical meanings.



(a) Parse tree from derivation (2.5). (b) Parse tree from derivation (2.6).

Figure 3 – Visual representation of derivations (2.5) and (2.6) in its parse tree form.

Compilers use parse trees to derive meaning and, therefore, ambiguous grammars are problematic for compiling (APPEL; PALSBERG, 2003). If we were to use some parsing generator algorithm, such as *LL*, *LR* and their variants, an effort should be made to transform such ambiguous grammars into unambiguous grammars (APPEL; PALSBERG, 2003). As an example, we can present an unambiguous grammar relative to the grammar (2.4) as

$$\begin{aligned}
\texttt{Expr} ::= {} & \texttt{Expr}\ \text{`}\texttt{+}\text{'}\ \texttt{Term} \\
\mid {} & \texttt{Expr}\ \text{`}\texttt{-}\text{'}\ \texttt{Term} \\
\mid {} & \texttt{Term} \\
\texttt{Term} ::= {} & \texttt{Term}\ \text{`}\texttt{*}\text{'}\ \texttt{Factor} \\
\mid {} & \texttt{Term}\ \text{`}\texttt{/}\text{'}\ \texttt{Factor} \\
\texttt{Factor} ::= {} & \text{`}\texttt{(}\text{'}\ \texttt{Expr}\ \text{`}\texttt{)}\text{'} \\
\mid {} & \text{`}\texttt{number}\text{'} \\
\mid {} & \text{`}\texttt{id}\text{'}
\end{aligned} \tag{2.7}$$

which, by adding new variables to the grammar, can express unambiguously in its parse trees that: the operators `*` and `/` *binds tighter*, or have a *higher precedence*, then `+` and `-`; and, the operators of the same *binding power*, or *precedence*, are left associative.

2.3.2 Review of top-down and bottom-up parsers

There are two main categories of parser. First we have parsers that try to construct the parse tree by finding a leftmost derivation starting from the root and creating the nodes of the parse three in preorder, called *top-down* parsers (AHO et al., 2006). And second we have parsers that try to construct the parse tree beginning at the leaves and working up to the root by performing a rightmost derivation in reverse applying a series of reductions on the input stream (AHO et al., 2006).

Top-down parsers, also called recursive descent parsers, have the advantage of its algorithms being simple enough to be used to construct parsers by hand (APPEL; PALSBERG, 2003). Although simple, classic implementations of recursive descent parsers, such as the *LL*(1) and *LL*(k) predictive parsers, cannot deal with ambiguous grammars and, since they rely on leftmost derivations, cannot deal with left recursions on grammar productions (AHO et al., 2006). There may also be the need to run a left factoring algorithm in order to generate a correct predictive parsing table for *LL*(1) and *LL*(k) automatic parser generators (AHO et al., 2006).

A non left recursive, left factored grammar for grammar (2.7) can be built as show in Figure 4. This grammar complies with every constraint imposed by a classic implementation of a predictive recursive descent parser since it is unambiguous, free of left recursions and left factored. An example implementation of predictive parsers

$$
\begin{aligned}
\texttt{Expr} &::= \texttt{Term Expr2} \tag{2.8}\\
\texttt{Expr2} &::= \text{‘}\texttt{+}\text{’}\ \texttt{Term Expr2} \tag{2.9}\\
&\mid \text{‘}\texttt{-}\text{’}\ \texttt{Term Expr2} \tag{2.10}\\
&\mid \varepsilon \tag{2.11}\\
\texttt{Term} &::= \texttt{Factor Term2} \tag{2.12}\\
\texttt{Term2} &::= \text{‘}\star\text{’}\ \texttt{Factor Term2} \tag{2.13}\\
&\mid \text{‘}\texttt{/}\text{’}\ \texttt{Factor Term2} \tag{2.14}\\
&\mid \varepsilon \tag{2.15}\\
\texttt{Factor} &::= \text{‘(’}\ \texttt{Expr}\ \text{‘)’} \tag{2.16}\\
&\mid \text{‘}\texttt{number}\text{’} \tag{2.17}\\
&\mid \text{‘}\texttt{id}\text{’} \tag{2.18}
\end{aligned}
$$

Figure 4 – A context-free grammar after left factoring and removing left recursions from grammar (2.7).

for the variables **`Expr`**, **`Expr2`**, **`Term`**, **`Term2`** and **`Factor`** can be found respectively at Figures 5a, 5b, 6a, 6b and 7.

```
fn parse_Expr(p: Parser) -> Parse {
  let t = parse_Term(p);
  let e2 = parse_Expr2(p);
  Expr(t, e2)
}
```

(a)

```
fn parse_Expr2(p: Parser) -> Parse {
  if p.at("+") | p.at("-") {
    let op = p.eat_token();
    let t = parse_Term(p);
    let e2 = parse_Expr2(p);
    Expr2(op, t, e2)
  } else {
    Empty
  }
}
```

(b)

Figure 5 – Predictive parsers for variables `Expr` (a), and `Expr2` (b), from examples (2.8) and (2.9).

```
fn parse_Term(p: Parser) -> Parse {
  let f = parse_Factor(p);
  let t2 = parse_Term2(p);
  Term(f, t2)
}
```

(a)

```
fn parse_Term2(p: Parser) -> Parse {
  if p.at("*") | p.at("/") {
    let op = p.eat_token();
    let f = parse_Factor(p);
    let t2 = parse_Term2(p);
    Term2(op, f, t2)
  } else {
    Empty
  }
}
```

(b)

Figure 6 – Predictive parsers for variables `Term` (a), and `Term2` (b), from examples (2.12) and (2.13).

We were once again forced to add new variables to the already unambiguous grammar (2.7). The transformations added variables without clear meanings, such as `Expr2` and `Term2`, further complicating the parse tree. In order to enrich the

```
fn parse_Factor(p: Parser) -> Parse {
  if p.at("(") {
    p.eat_token();
    let expr = parse_Expr(p);
    p.expect(")");
    NestedExpr(expr)
  } else if p.at("id") {
    let token = p.eat_token();
    Id(token)
  } else if p.at("number") {
    let token = p.eat_token();
    Number(token)
  } else {
    panic!("Parse error.");
  }
}
```

Figure 7 – A predictive parser for variable `Factor` in (2.16).

capabilities of recursive descendant parsers, advances were made to allow recursive descendant parsers to parse some ambiguous grammars with left recursion without the need to left factoring. Pratt (1973) proposed a *top-down operator precedence* approach to parse arithmetic expressions by assigning a total order to tokens using a function for the left and right binding power of tokens in order to uniquely create parse trees of ambiguous left recursive expression grammars. An example Pratt parser for grammar (2.4) is found at Figure 8. Pratt parsers work by combining recursion with iteration. If we try to parse the sentence $\texttt{id}_1$ `+` `number` `*` $\texttt{id}_2$ it:

- parses $\texttt{id}_1$ and name its node 'left';
- enters the loop and parse `+` which have a binding power to the left of 1 and to the right of 2;
- checks if it binds to the left stronger than the minimum binding power, in this case it does not since the minimum starts at 0, continuing on to recursively parse '`number`' which its node becomes the new 'left' in the new recursion step;
- the next operator is then `*` with binding power 3 and 4, this time the left binding power still is not smaller than the minimum of 2, and we recursively parse $\texttt{id}_2$ which again its node becomes a new 'left';
- now, upon trying to parse a new operator, we break from the loop and return the node for $\texttt{id}_2$ and name it 'right'; then, we construct a node `Expr` with 'left', which is `number` for this iteration, the operator `*`, and 'right', which is $\texttt{id}_2$;
- we then return again with another `Expr` node with 'left', which now is $\texttt{id}_1$, the operator `+`, and the last returned expression as 'right' finally forming the parse tree with correct precedence found at Figure 3b.

```
fn parse_Expr(p: Parser) -> Parse {
  parse_Expr_binding_power(p, 0)
}
fn parse_Expr_binding_power(p: Parser, min_bp: Int) -> Parse {
  let left = if p.at("id") {
    let token = p.eat_token();
    Id(token)
  } else if p.at("number") {
    let token = p.eat_token();
    Number(token)
  } else if p.at("(") {
    p.eat_token();
    let expr = p.parse_Expr();
    p.expect(")");
    expr
  } else {
    panic!("Parse error.");
  };
  loop {
    let op = if p.at("+") { Sum }
      else if p.at("-") { Minus }
      else if p.at("*") { Times }
      else if p.at("/") { Div }
      else { break; }
    let (left_bp, right_bp) = binding_power(op);
    if left_bp < min_bp {
      break;
    }
    p.eat_token();
    let right = parse_Expr_binding_power(p, right_bp);
    left = Expr(left, op, right);
  }
  left
}
fn binding_power(op: BinOp) -> (Int, Int) {
  match op {
    Sum | Minus => (1, 2),
    Times | Div => (3, 4),
  }
}
```

Figure 8 – A Pratt parser for the ambiguous expression grammar (2.4).

The loop to keep checking if the next operation binds stronger than the minimum for the recursion guaranties both correct associativity and operator precedence (PRATT, 1973).

Besides the work of Pratt (1973), there are recursive descent parsers which can parse a bigger range of ambiguous and left recursive grammars. Frost, Hafiz, and Callaghan (2008) proposes a handwritten parser combinator approach that uses composable higher-order functions, together with memoization and backtracking, in order to allow parsing both ambiguity and left recursive grammars in polynomial time. There is also the works of Ford (2002) that proposes a handwritten parser approach, called *Packrat*, using memoization, backtracking, and unlimited look ahead to parse *LL*(k) and *LR*(k) grammars by using *Parsing Expression Grammar* (PEG) definitions, formalized by Ford (2004), which differs from context-free grammars by imposing a total order to the rules set of the grammar. Therefore, PEGs cannot be ambiguous because there is no choice involved in which production to choose for derivation (FORD, 2004). Furthermore, Warth, Douglass, and Millstein (2008) extended Packrat to allow left-recursion by changing its memoization process.

Bottom-up parsers have fewer context-free grammar constraints (AHO et al., 2006). However, they still have troubles with ambiguous grammars due to shift-reduce conflicts but no longer require a non-left recursive and left factored grammar (AHO et al., 2006). Writing a bottom-up shift-reduce parser for an LR class grammar is not an easy task, thankfully parser generators for $LR(1)$ and $LR(k)$ languages are capable of generating very efficient parsers for a large range of context-free grammars (AHO et al., 2006). Even allowing grammar designers to solve shifting-reduce conflicts by hand assigning whether to shift or to reduce for every conflict (AHO et al., 2006).

Although using bottom-up parser generators are desirable for its expressiveness and efficiency, in the works of Kladov (2020a,b) for the Rust Analyzer, and Parr (2013) for the ANTLR parser generator, the authors advocate to implement handwritten recursive descent top-down parser for it allows the implementation of better error messages and error recovery algorithms for modern compiler implementations and tools.

2.3.3 Semantic actions and syntax-directed translations

As put by Appel and Palsberg (2003) "A compiler must do more than recognize whether a sentence belongs to the language of a grammar, it must do something useful with that sentence". The specification of *semantic actions* allows us to do something useful with sentences that are parsed (APPEL; PALSBERG, 2003). They are pieces of code that we can attach to the body of grammar production rules to specify *syntax-directed definitions* (AHO et al., 2006). Syntax-directed definitions, then, are used to perform *syntax-directed translations* by executing the code specified by the semantic actions during specific moments of the parsing process (AHO et al., 2006). Take the following syntax-directed definition

$$\texttt{Expr} ::= \texttt{Term}\ \texttt{Expr2}\ \texttt{\{Expr.n = Expr(Term.n, Expr2.n)\}} \tag{2.19}$$

Note that the attribute `n` for the variables `Expr`, `Term` and `Expr2`, can be thought of as the return node value of a parsing function for this production. If we give a closer look at the parser implemented in Figure 5a, we can see this semantic action at line 4, after parsing `Term` and `Expr2`, returning the value of the attribute `Expr.n`. Furthermore, the position of the semantic action in a production body determines when the semantic action executes during parsing. In this example it is positioned after the parser finishes parsing the production.

In Appel and Palsberg (2003) the autors state that it is possible to write an entire compiler into the semantic actions of a parser. However, this approach limits the compiler to analyze the source program in the strict order it is parsed (APPEL; PALSBERG, 2003). Hence, using some sort of intermediate representation between syntax analysis and semantics analysis allows the semantic analyzer to be free from the constraints of the parsing algorithm (APPEL; PALSBERG, 2003). One possible solution is to design the semantic actions in such a way that they output an intermediate tree data structure (APPEL; PALSBERG, 2003). When the parser builds a tree with leaf nodes for each token and interior nodes for each production parsed we call such tree a *concrete parse tree* (APPEL; PALSBERG, 2003).

An example of a concrete parse tree created by the semantic actions of a parser implementing the grammar presented in Figure 4 can be found in Figure 9.

```
Expr
  Term
    Factor
      Number "5"
  Expr2
    Plus "+"
    Term
      Factor
        Id "a"
```

Figure 9 – A sample concrete syntax tree for the grammar in Figure 4 representing sentence `5 + a`.

2.3.4 Abstract syntax trees

Sometimes a concrete parse tree is too verbose, like a concrete tree for the grammar in Figure 4, and, it is ideal to transform this tree into a simpler one in order to facilitate semantic analysis (APPEL; PALSBERG, 2003). We can then create a second grammar with the *abstract syntax* of the language. The abstract syntax grammar does not need to follow the constraints of syntax analysis and only exists to facilitate the later stages of compilation (APPEL; PALSBERG, 2003). For example, we can use the ambiguous grammar at (2.4) as the abstract syntax for the language described by the grammar in Figure 4.

A compiler could easily translate a concrete parse tree into an *Abstract Syntax Tree* (AST) (APPEL; PALSBERG, 2003). In order to build tree translators, compiler implementation use a variety of programming tools in order to traverse the input tree and output the desired tree. Appel and Palsberg (2003) explains how to use the visitor programming pattern in object-oriented programming languages with inheritance and interfaces in order to create tree traversal algorithms. Furthermore, Pierce (2002) advocates for the use of programming languages with pattern matching and abstract data types in order to define recursive functions in a functional programming fashion to traverse the tree.

A tree translator implementation using recursive functions and pattern matching for the concrete parse tree `Parse` defined in Figure 10a created by the predictive parser in Figures 6 and 7 into the AST defined in Figure 10b can be found in Figures 11 to 13.

```
enum Parse {
  Expr(Parse, Parse),
  Expr2(Token, Parse, Parse),
  Term(Parse, Parse),
  Term2(Token, Parse, Parse),
  Number(Token),
  Id(Token),
  Empty,
}
```

(a) Concrete syntax tree definition for the predictive parser in Figures 6 and 7.

```
enum Ast {
  Expr(Ast, BinOp, Ast),
  Number(Int),
  Id(String),
}
enum BinOp {
  Plus,
  Minus,
  Times,
  Div,
}
```

(b) Abstract syntax tree definition.

Figure 10 – Data structures for a concrete parse tree and an equivalent abstract syntax tree.

```
fn translate_expr(expr: Parse) -> Ast {
  match expr {
    Parse::Expr(term, cont) => {
      let ast_term = translate_term(term);
      match translate_expr2(cont) {
        None => ast_term,
        Some((op, ast_cont)) => Ast::Expr(ast_term, op, ast_cont),
      }
    }
    _ => panic!("not an Expr"),
  }
}
fn translate_expr2(expr2: Parse) -> Option<(BinOp, Ast)> {
  match expr2 {
    Parse::Expr2(op_tok, term, cont) => {
      let op = into_binop(op_tok);
      let ast_term = translate_term(term);
      let ast = match translate_expr2(cont) {
        None => (op, ast_term),
        Some(op2, ast_term2) => (op, Ast::Expr(ast_term, op2, ast_term2)),
      };
      Some(ast)
    }
    Parse::Empty => None,
    _ => panic!("not an Expr2"),
  }
}
```

Figure 11 – A tree transformation from the `Expr` and `Expr2` nodes in Figure 10a into the AST nodes in Figure 10b.

```
fn translate_term(term: Parse) -> Ast {
  match term {
    Parse::Term(factor, cont) => {
      let ast_factor = translate_factor(factor);
      match translate_factor2(cont) {
        None => ast_factor,
        Some((op, ast_cont)) => Ast::Expr(ast, op, ast_cont),
      }
    }
    _ => panic!("not an Term"),
  }
}
fn translate_term2(term2: Parse) -> Option<(BinOp, Ast)> {
  match term2 {
    Parse::Term2(op_tok, factor, cont) => {
      let op = into_binop(op_tok);
      let ast_factor = translate_factor(factor);
      let ast = match translate_term2(cont) {
        None => (op, ast_factor),
        Some(op2, ast_factor2) => {
          (op, Ast::Expr(ast_factor, op2, ast_factor2))
        }
      };
      Some(ast)
    }
    Parse::Empty => None,
    _ => panic!("not an Term2"),
  }
}
```

Figure 12 – A tree transformation from the `Term` and `Term2` nodes in Figure 10a into the Ast nodes in Figure 10b.

```
fn translate_factor(factor: Parse) -> Ast {
  match factor {
    Parse::NestedExpr(expr) => translate_expr(expr),
    Parse::Number(token) => Ast::Number(token.into_int()),
    Parse::Id(token) => Ast::Id(token.into_string()),
    _ => panic!("not an Factor"),
  }
}
```

Figure 13 – A tree transformation from the `NestedExpr`, `Number`, and `Id` nodes in Figure 10a into the Ast nodes in Figure 10b.

2.3.5 Syntax error recovery

When a parser fails to find a derivation for the input it is called a syntax error (APPEL; PALSBERG, 2003). If we find an error, it would be advantageous for the user of the compiler if the parsing did not halt the compilers' execution on the first error (APPEL; PALSBERG, 2003). This is what can be accomplished with error recovery (APPEL; PALSBERG, 2003). One way we can build error recovery into top-down parsers is by the use of recovery token sets. Essentially, once a parser using recovery token set finds an error, it discards all tokens until it finds a token in the recovery token set, then returns an error node containing the discarded tokens hopping that the father nodes will be able to continue parsing from the token found in the recovery token set.

2.4 SEMANTIC ANALYSIS

The third phase of a compiler is called semantic analysis. It is responsible to use the AST generated by the parser to check if the source program is semantically consistent with the language specification. An important part of semantic analysis is *type checking* where it will try to validate the program to the language specification's type system.

If we feed a type checker with the abstract syntax tree for the program

```
b + 60 / c
```

it may make use of a *context* containing the type information of the identifiers `b` and `c` to type check all nodes of the tree for consistency. Suppose a context that maps the identifier `c` to the type `Bool`, written `c : Bool`, which would compose of the values `true` and `false`. With this information, the type checker would be able to decide if the operation `60 / c` is semantically sound to the language's type system. Does the language type system specifies as valid to divide a number by a boolean? If it does, what does it mean to divide the value `60` by `false`?

Maybe, whoever designed the language decided that, if the dividend is a value of type `Int` and the divisor is a value of type `Bool`, it will use some rule to convert the divisor's type to number in order to keep the program semantically sound. Maybe, the language's type system forbids this behavior and will halt the compilation process with some error message. These are decisions made when building a language's type system.

In order to star reasoning about the semantic aspects of a language suppose the following abstract syntax

$$\begin{aligned} \texttt{term} ::= \; & \texttt{true} \\ \mid \; & \texttt{false} \\ \mid \; & \texttt{if term then term else term} \\ \mid \; & \texttt{0} \\ \mid \; & \texttt{succ term} \\ \mid \; & \texttt{pred term} \\ \mid \; & \texttt{iszero term} \end{aligned} \tag{2.20}$$

Note that the grammar for an abstract tree does not have different notations for its terminal and non-terminal symbols in order to not clutter the inference rules with notation symbols (PIERCE, 2002). The symbol `term` at the left of ::= defines the set of *terms* and defines that we are going to use the symbol `term` to range over the set terms (PIERCE, 2002). The same symbol `term` when used at the right of ::= is called a *metavariable* simply to differentiate from the variables of the programming language in question, and, we may substitute the `term` metavariables for any instance of terms (PIERCE, 2002).

The set of productions (2.20) defines a set of all possible terms, although compact and expressive it is only one of several ways of describing the syntax of a language (PIERCE, 2002). According to Pierce (2002), using the grammar to define the set of all terms is actually just a compact notation for the following inductive definition: The set of all terms is the smallest set T such that

$$\{\texttt{true},\ \texttt{false},\ \texttt{0}\} \subseteq T \tag{2.21}$$

$$\text{if } \texttt{term} \in T\text{, then } \{\texttt{succ term},\ \texttt{pred term},\ \texttt{iszero term}\} \subseteq T \tag{2.22}$$

$$\text{if } \texttt{term}_1, \texttt{term}_2, \texttt{term}_3 \in T\text{, then } \texttt{if term}_1 \texttt{ then term}_2 \texttt{ else term}_3 \in T \tag{2.23}$$

Formally, this definition defines T as a set of *trees* and not a set of strings as we show in Section 2.3 with derivations (PIERCE, 2002). A shorthand definition for the same inductive definition of terms can be given in *inference rules*, as following:

$$\texttt{true} \in T \qquad \texttt{false} \in T \qquad \texttt{0} \in T$$

$$\frac{\texttt{term} \in T}{\texttt{succ term} \in T} \qquad \frac{\texttt{term} \in T}{\texttt{pred term} \in T} \qquad \frac{\texttt{term} \in T}{\texttt{pred term} \in T}$$

$$\frac{\texttt{term}_1 \in T \qquad \texttt{term}_2 \in T \qquad \texttt{term}_3 \in T}{\texttt{if term}_1 \texttt{ then term}_2 \texttt{ else term}_3 \in T}$$

where the first three rules restate the first clause in (2.21), the middle three rules restate the second clause in (2.22), and the last rule restate clause in (2.23) (PIERCE, 2002). Each rule means that if we found the statements in the premises listed above the line, then we may derive the conclusion below the line (PIERCE, 2002). When defining the syntax as inference rules, the fact that T is the smallest possible set is not stated explicitly but implied, also, the rules with no premise are called *axioms* and are written with no bar, since there is nothing to put above it (PIERCE, 2002). Furthermore, Pierce (2002) points out that what we are calling *inference rules* are actually *rule schemas* which represents the infinite set of *concrete rules* that can be obtained by substituting each variable by all possible sentences of the syntactic category, e.g., substituting each metavariable for `term` by every possible term in the inference rules above.

We can than proof that

$$\texttt{if true than succ 0 else pred succ 0}$$

is a `term` as follows:

$$\frac{\texttt{true} \in T \qquad \dfrac{\texttt{0} \in T}{\texttt{succ 0} \in T} \qquad \dfrac{\dfrac{\texttt{0} \in T}{\texttt{succ 0} \in T}}{\texttt{pred succ 0} \in T}}{\texttt{if true than succ 0 else pred succ 0} \in T}$$

Here we see the concrete inference rules in play, and not the rules schemas. We constructed the inference rules in such a way that proving a tree is part of T is no different from finding a derivation for a string s for the grammar.

2.4.1 Operational Semantics

The *operational semantics* of a language allows for the precise definition of how terms are evaluated, that is, the precise definition of the *semantics* of the language (PIERCE, 2002). It specifies the behavior of a programming language by defining an *abstract machine* that uses the terms of the programming language as its instructions instead of a low-level processor's instruction set (PIERCE, 2002).

For simple languages like (2.20) the state of the abstract machine can be defined by just a `term` and its behavior by a *transition function* that for each `term` either performs an evaluation step, or halts and announces that the evaluation has ended (PIERCE, 2002). In order to define operational semantics for (2.20) we first extend the abstract syntax with a new metavariable `value` for the possible end results of evaluation as

$$\begin{array}{rcl} \texttt{value} & ::= & \texttt{true} \\ & | & \texttt{false} \\ & | & \texttt{nvalue} \\ \texttt{nvalue} & ::= & \texttt{0} \\ & | & \texttt{succ nvalue} \end{array} \tag{2.24}$$

and second, we define and *evaluation relation*, or *judgments*, on terms, written

$$\texttt{term} \rightarrow \texttt{term}_1$$

meaning `term` evaluates to $\texttt{term}_1$ in one step (PIERCE, 2002).

The evaluation relation for (2.20) is the smallest relation defined by the inference rules in Figure 14. Using the evaluation relations from Figure 14 we can evaluate the following program

$$\texttt{if iszero 0 then succ 0 else false} \tag{2.25}$$

$$\frac{}{\texttt{if true then term}_1\ \texttt{else term}_2 \rightarrow \texttt{term}_1}\ \text{Eval-IfTrue}$$

$$\frac{}{\texttt{if false then term}_1\ \texttt{else term}_2 \rightarrow \texttt{term}_2}\ \text{Eval-IfFalse}$$

$$\frac{\texttt{term}_1 \rightarrow \texttt{term}_4}{\texttt{if term}_1\ \texttt{then term}_2\ \texttt{else term}_3 \rightarrow \texttt{if term}_4\ \texttt{then term}_2\ \texttt{else term}_3}\ \text{Eval-If}$$

$$\frac{\texttt{term}_1 \rightarrow \texttt{term}_2}{\texttt{succ term}_1 \rightarrow \texttt{succ term}_2}\ \text{Eval-Succ}$$

$$\frac{}{\texttt{pred succ nvalue} \rightarrow \texttt{nvalue}}\ \text{Eval-PredSucc}$$

$$\frac{\texttt{term}_1 \rightarrow \texttt{term}_2}{\texttt{pred term}_1 \rightarrow \texttt{pred term}_2}\ \text{Eval-Pred}$$

$$\frac{}{\texttt{iszero 0} \rightarrow \texttt{true}}\ \text{Eval-IszeroZero}$$

$$\frac{\texttt{term}_1 \rightarrow \texttt{term}_2}{\texttt{iszero term}_1 \rightarrow \texttt{iszero term}_2}\ \text{Eval-Iszero}$$

Figure 14 – The evaluation relations for the operational semantics of the abstract syntax (2.20) with values defined by (2.24). Adapted from Pierce (2002).

By looking at Figure 14, we see that the only inference rule we can apply is Eval-If. When applying this rule to (2.25) we get

$$\frac{\dfrac{}{\texttt{iszero 0} \rightarrow \texttt{true}}\ \text{Eval-IszeroZero}}{\texttt{if iszero 0 then succ 0 else false} \rightarrow \texttt{if true then succ 0 else false}}\ \text{Eval-If}$$

proving that

$$\texttt{if iszero 0 then succ 0 else false} \rightarrow \texttt{if true then succ 0 else false}$$

by first applying rule Eval-If, then applying the axiom Eval-IszeroZero. We can continue the evaluation by applying rule Eval-IfTrue as in

$$\frac{}{\texttt{if true then succ 0 else false} \rightarrow \texttt{succ 0}}\ \text{Eval-IfTrue} \tag{2.26}$$

thus proving that

$$\texttt{if true then succ 0 else false} \rightarrow \texttt{succ 0}$$

Now the only evaluation rule that fits `succ 0` is Eval-Succ, but if we try to apply it like in

$$\frac{\dfrac{}{\texttt{0} \to ?}\,?}{\texttt{succ 0} \to ?}\,\text{Eval-Succ}$$

we quickly find that there is no rule to evaluate `0`, so the abstract machine halts and checks if the last evaluated result is a value (PIERCE, 2002). In this case `succ 0` does belong in `values`, so the abstract machine successfully evaluated

$$\texttt{if iszero 0 then succ 0 else false} \to \texttt{succ 0}$$

Note that if the `if` condition evaluated to `false` instead of `true`, we would have applied rule Eval-IfFalse instead of Eval-IfTrue at evaluation step (2.26). Thus, evaluating the input term to `false` instead of `succ 0` which are values of different *kinds*, but this behavior was intentionally described by the evaluation relation rules.

If, for example, the programmer of this language wrote a program that at some point evaluated to

`if 0 then true else false`

or

`succ true`

we quickly find out that there is no evaluation rule that can be applied, and, given that those terms are not present in `value`, the abstract machine not only halts, but we say it is *stuck* (PIERCE, 2002). In this case we would either have to define what it means for `0` to be used as a condition and what it means for constructing the successor of `true`, or halt the execution and return a runtime error to the user of the program being executed (PIERCE, 2002).

Terms that get stuck during evaluation correspond to meaningless or erroneous programs (PIERCE, 2002). We would rather be able to tell, without evaluating the program, that its evaluation will *not* get stuck (PIERCE, 2002). For that, we would need to be able to differentiate between the different kinds of values computed by terms (PIERCE, 2002). In the next section we introduce to the language (2.20) the `Nat` and `Bool` types in order to make sure every program evaluated does not get stuck.

2.4.2 The typing relation

In order to ensure that programs will correctly evaluate to a value, we will introduce a new relation called the *typing relation*.

The typing relation for arithmetic expressions is written as `term : Type`, and means that `term` is of type `Type` where `Type` is a new syntactic form for the types of the language (PIERCE, 2002). Also, the relation is defined by a set of inference rules assigning types to terms (PIERCE, 2002).

We introduce the syntactic form for the metavariable `Type` to the syntax (2.20) as

$$\begin{aligned}\texttt{Type} ::= {} & \texttt{Nat}\\ \mid {} & \texttt{Bool}\end{aligned}\tag{2.27}$$

and define the inference rules for the typing relation of the abstract syntax (2.20) as show in Figure 15.

$$\frac{}{\texttt{true : Bool}}\;\text{Ty-True} \qquad\qquad \frac{}{\texttt{false : Bool}}\;\text{Ty-False}$$

$$\frac{\texttt{term}_1\texttt{ : Bool} \quad \texttt{term}_2\texttt{ : Type} \quad \texttt{term}_3\texttt{ : Type}}{\texttt{if term}_1\texttt{ then term}_2\texttt{ else term}_3\texttt{ : Type}}\;\text{Ty-If}$$

$$\frac{\texttt{t}_1\texttt{ : Nat}}{\texttt{succ t}_1\texttt{ : Nat}}\;\text{Ty-Succ} \qquad \frac{\texttt{t}_1\texttt{ : Nat}}{\texttt{pred t}_1\texttt{ : Nat}}\;\text{Ty-Pred} \qquad \frac{\texttt{t}_1\texttt{ : Nat}}{\texttt{iszero t}_1\texttt{ : Bool}}\;\text{Ty-Iszero}$$

Figure 15 – The inference rules for the typing relation of the abstract syntax (2.20) with types defined by (2.27). Adapted from Pierce (2002).

Now we can use the inference rules from Figure 15 to prove if the term

$$\texttt{iszero succ 0 : Bool}$$

by applying the following sequence of inferences

$$\dfrac{\dfrac{\dfrac{}{\texttt{0 : Nat}}\;\text{Ty-Zero}}{\texttt{succ 0 : Nat}}\;\text{Ty-Succ}}{\texttt{iszero succ 0 : Bool}}\;\text{Ty-IsZero}$$

but, if we try to prove something wrong like

$$\texttt{succ true : Num} \quad \text{or} \quad \texttt{if 0 then true else false : Bool}$$

we will quickly find that there are no rules that allow us to reach the type relation axioms. Furthermore, if we try to prove and infer the type of the example (2.25) we end up with the following rule applications

$$\dfrac{\texttt{iszero 0 : Bool} \qquad \dfrac{\dfrac{}{\texttt{0 : Nat}}\;\text{Ty-Zero}}{\texttt{succ 0 : Nat}}\;\text{Ty-Succ} \qquad \texttt{false : Bool}}{\texttt{if iszero 0 the succ 0 else false : ?}}\;\text{Ty-If}$$

when trying to inductively define the typing relation for example (2.25), we can not properly use the induction rule Ty-If because the terms in both branches of the `if` evaluate to different types, and the rule Ty-If mandates that both branches evaluate to the same type for the rule to be sated.

2.4.3 Pure simply typed lambda calculus

In this section we will be discussing a variation of what Pierce (2002) calls *Pure simply typed lambda-calculus*.

To bring our discussion to the simply typed lambda calculus, we resume all constructs we discussed so far relevant to type checking, together with the newly added ones as the following abstract syntax

$$\begin{aligned}
\texttt{term} ::= \;& \texttt{true} \\
\mid\;& \texttt{false} \\
\mid\;& \texttt{if term then term else term} \\
\mid\;& \texttt{0} \\
\mid\;& \texttt{succ term} \\
\mid\;& \texttt{pred term} \\
\mid\;& \texttt{iszero term} \\
\mid\;& \texttt{var} \\
\mid\;& \texttt{let var = term in term} \\
\mid\;& \lambda\texttt{var : Type.term} \\
\mid\;& \texttt{term term} \\
\texttt{var} ::= \;& \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \texttt{aa} \mid \texttt{ab} \mid \ldots \\
\texttt{Type} ::= \;& \texttt{Bool} \\
\mid\;& \texttt{Nat} \\
\mid\;& \texttt{Type} \to \texttt{Type}
\end{aligned} \tag{2.28}$$

But what happens if we try to prove `succ a : Nat` for example? We currently have no rules that allow us to infer the type of the variable `a`. Nor do we have the capabilities with the current presented theory. In order to be capable of expressing the types of variables we need to extend the typing relation from a two-place relation into a three-place relation adding a context Γ (PIERCE, 2002).

The context Γ, also called *typing context*, is a set of assumptions of the form `var : Type` (PIERCE, 2002). It is described by the syntactic form

$$\begin{aligned}
\Gamma ::= \;& \Gamma, \texttt{var : Type} \\
\mid\;& \emptyset
\end{aligned} \tag{2.29}$$

Γ is essentially a list of typing relations that grows to the right by applying the `,` operator, and we can also take away a typing relation from the context clever use of this notation during the design of inference rules (PIERCE, 2002).

The new three-place typing relation is written $\Gamma \vdash \texttt{term : Type}$, and it means that `term` has type `Type` under the context Γ (PIERCE, 2002). If the context is empty we write $\emptyset \vdash \texttt{term : Type}$, but the $\emptyset$ is usually omitted as in $\vdash \texttt{term : Type}$.

Pierce (2002) defines the three-place typing relation inference rules as show in Figure 16. Supposing all typing relation rules from Figure 15 were updated to the newer three-part type relation, In order to exemplify the workings of rules Ty-Var, Ty-Le, Ty-FnAbs, and Ty-FnApp, we will build the inference tree for the term

$$\frac{\texttt{var : Type} \in \Gamma}{\Gamma \vdash \texttt{var : Type}}\ \text{Ty-Var}$$

$$\frac{\Gamma \vdash \texttt{term}_1\ \texttt{: Type}_1 \qquad \Gamma\texttt{, var : Type}_1 \vdash \texttt{term}_2\ \texttt{: Type}_2}{\Gamma \vdash \texttt{let var = term}_1\ \texttt{in term}_2\ \texttt{: Type}_2}\ \text{Ty-Let}$$

$$\frac{\Gamma\texttt{, var : Type}_1 \vdash \texttt{term : Type}_2}{\Gamma \vdash \lambda\texttt{var : Type}_1\texttt{.term : Type}_1 \to \texttt{Type}_2}\ \text{Ty-FnAbs}$$

$$\frac{\Gamma \vdash \texttt{term}_1\ \texttt{: Type}_1 \to \texttt{Type}_2 \qquad \Gamma \vdash \texttt{term}_2\ \texttt{: Type}_1}{\Gamma \vdash \texttt{term}_1\ \texttt{term}_2\ \texttt{: Type}_2}\ \text{Ty-FnApp}$$

Figure 16 – The inference rules for the three-place typing relation of the abstract syntax (2.28). Adapted from Pierce (2002).

$$\texttt{let fun = }\lambda\texttt{x : Nat.iszero x in fun succ 0}$$

as presented in the following inference trees

$$\cfrac{\cfrac{\cfrac{\texttt{x : Nat} \in \Gamma}{\Gamma\texttt{, x : Nat} \vdash \texttt{x : Nat}}\ \text{Ty-Var}}{\Gamma\texttt{, x : Nat} \vdash \texttt{iszero x : Bool}}\ \text{Ty-Iszero}}{\Gamma \vdash \lambda\texttt{x : Nat.iszero x : Nat} \to \texttt{Bool}}\ \text{Ty-FnAbs}$$

$$\frac{\texttt{fun : Nat} \to \texttt{Nat} \in \Gamma}{\Gamma\texttt{, fun : Nat} \to \texttt{Nat} \vdash \texttt{fun : Nat} \to \texttt{Nat}}\ \text{Ty-Var}$$

$$\cfrac{\cfrac{}{\Gamma\texttt{, fun : Nat} \to \texttt{Nat} \vdash \texttt{0 : Nat} \in \Gamma}\ \text{Ty-Zero}}{\Gamma\texttt{, fun : Nat} \to \texttt{Nat} \vdash \texttt{succ 0 : Nat}}\ \text{Ty-Succ}$$

$$\frac{\Gamma\texttt{, fun : Nat} \to \texttt{Bool} \vdash \texttt{fun : Nat} \to \texttt{Bool} \qquad \Gamma\texttt{, fun : Nat} \to \texttt{Bool} \vdash \texttt{succ 0 : Nat}}{\Gamma\texttt{, fun : Nat} \to \texttt{Bool} \vdash \texttt{fun succ 0 : Bool}}\ \text{Ty-FnApp}$$

$$\frac{\Gamma \vdash \lambda\texttt{x : Nat.iszero x : Nat} \to \texttt{Bool} \qquad \Gamma\texttt{, fun : Nat} \to \texttt{Bool} \vdash \texttt{fun succ 0 : Bool}}{\Gamma \vdash \texttt{let fun = }\lambda\texttt{x : Nat.iszero x in fun succ 0 : Bool}}\ \text{Ty-Let}$$

2.4.4 A type checker implementation for the simply typed lambda calculus

The goal of a type checker is to show that a program can be correctly evaluated. In order to achieve this goal we introduced the typing relation and the inference rules that defines it. Now, for an algorithmic definition of a type checker, we take the typing relation

$$\Gamma \vdash \texttt{term : Type}$$

and, from the inversion lemma for the typing relation (PIERCE, 2002), interpret it as a function

$$\texttt{typeof(}\Gamma\texttt{, term)} = \texttt{Type}$$

We can easily construct Abstract Data Types (ADTs) for the abstract syntax of Γ, `term`, and `Type` presented in the simply typed lambda calculus defined by grammar (2.28) as show by Figure 17.

```rust
enum Context {
  Empty,
  Cons(
    Context,
    (String, Type)
  ),
}
```

(a)

```rust
enum Type {
  Num,
  Bool,
  Fn(Type, Type),
}
```

(b)

```rust
enum Term {
  True,
  False,
  If(Term, Term, Term),
  Zero,
  Succ(Term),
  Pred(Term),
  IsZero(Term),
  Var(String),
  Let(String, Term, Term),
  Fn(String, Type, Term),
  App(Term, Term),
}
```

(c)

Figure 17 – Abstract Data Types for the context Γ (a), `Type` (b), and `term` (c) from the abstract syntax defined in grammar (2.28) written in Rust.

Given the ADT definitions of `Context`, `Term`, and `Type`, we can now define the `typeof` function as show in Figure 18. The definition of `typeof` in Figure 18 is directly derived from the inference rules from Figure 16 and the inference rules from Figure 15 with added context propagation. Furthermore, from the inversion lemma of the typing relation (PIERCE, 2002), if the `typeof` function is able to output a valid type, then, the input term is consider to be well-typed and sound to the language's type system.

```rust
fn typeof(cx: Context, t: Term) -> Type {
  match t {
    True => Bool,
    False => Bool,
    If(t1, t2, t3) => if typeof(cx, t1) == Bool {
      let if_ty = typeof(cx, t2);
      if if_ty == typeof(cx, t3) {
        if_ty
      } else {
        panic!("if branches differ in type")
      }
    } else {
      panic!("if condition term not a Bool")
    },
    Zero => Type::Num,
    Succ(t) => if typeof(cx, t) == Type::Num {
      Type::Num
    } else {
      panic!("succ argument term not a Num")
    },
    Pred(t) => if typeof(cx, t) == Type::Num {
      Type::Num
    } else {
      panic!("pred argument term not a Num")
    },
    IsZero(t) => if typeof(cx, t) == Type::Num {
      Type::Bool
    } else {
      panic!("iszero argument term not a Num")
    }
    Var(x) => get_type_in_context(cx, x)
    Let(x, t1, t2) => {
      let x_ty = typeof(t1);
      let cx1 = add_binding(cx, (x, x_ty));
      typeof(cx1, t2)
    }
    Fn(x, x_ty, t) => {
      let cx1 = add_binding(cx, (x, x_ty));
      Type::Fn(ty, typeof(cx1, t))
    }
    App(t1, t2) => match typeof(cx, t1) {
      Fn(in_ty, out_ty) => if typeof(cx, t2) == in_ty {
        out_ty
      } else {
        panic!("Application argument term type differs from function's input type")
      },
      _ => panic!("Application function term not a Function type")
    }
  }
}
```

Figure 18 – An implementation of the `typeof` function for typechecking the simply typed lambda calculus described in Section 2.4.3 written in Rust.

2.5 INTERMEDIATE CODE GENERATION

As the last phase of a compiler's front end we have the intermediate code generation. The intermediate code generation will take as input the abstract syntax tree generated from semantic analysis and output an intermediate representation (IR). In a sense, we will be evaluating the source AST but, instead of generating the resulting value of the AST's computation as we formalized in Section 2.4.1 using operational semantics, we will be evaluating the source AST into another syntactical representation.

Compilers may construct a sequence of many intermediate syntactical representations until reaching the target machine executable code where each IR is suitable for the kind of analysis and operations it is executed upon (AHO et al., 2006). We have already described how to evaluate a concrete parse tree into an AST in Section 2.3.4, and, the process of creating tree traversals algorithms in order to perform tree transformations continues into evaluating an AST into an IR suitable for code optimization and target code generation.

A suitable IR has several qualities:

- It must be convenient for the analysis phase to translate an AST into (APPEL; PALSBERG, 2003);
- It must be convenient for a back-end to translate it into machine code for the desired range of target machines (APPEL; PALSBERG, 2003);
- The syntactical constructs of the IR must have clear and simple meanings allowing for the specification and implementation of optimizing transformations of the IR (APPEL; PALSBERG, 2003).

Thus, a good IR for code generation would abstract the desired range of target machines into an abstract machine that is capable of describing target machine operations either directly or by using the abstract machine operations as building blocks for more complex operations (APPEL; PALSBERG, 2003). This can be achieved by the IR represented by the grammar in Figure 19.

2.5.1 Three-address code

The three-address code IR is a linearized representation of a syntax tree (AHO et al., 2006). For example, the source AST for

```
let x = 1 + 2 * 3 in x / -x
```

```
      IR ::= stmtList                     expr ::= const
stmtList ::= stmt stmtList | stmt                | label
    stmt ::= expr                                | name
           | name = expr                         | expr binop expr
           | store name expr                     | unop expr
           | jump expr                           | call expr exprList
           | cjump expr expr                     | load expr
           | label stmt                 exprList ::= expr exprList | ε
```

Figure 19 – An intermediate representation for code generation. Adapted from Appel and Palsberg (2003).

could be linearized as the following sequence of three-address code assignment instructions:

```
t1 = 2 * 3
 x = 1 + t1
t2 = 0 - x
t3 = x / t2
```

Each three-address code instruction is formed by *addresses* and at most one operation (AHO et al., 2006). An address can be one of the following:

- A *constant* `c` which are values for the primitive types supported by the operations of an abstract machine.
- A *name* `n` which is can either be generated from the variables in the source AST, or, an automatically generated temporary name.

where we find a singe assigned address at the instruction's left side and on the right side we have, either a single address, or, an n-ary operator together with n address arguments (AHO et al., 2006).

2.5.2 Single Static Assignment (SSA)

The Single Static Assignment (SSA) is another intermediate representation (IR) form, but unlike the three-address code, it's distinguished by its property wherein every variable is assigned precisely once (CYTRON et al., 1991). This unique characteristic of SSA form has significant implications for the subsequent stages of compilation.

For instance, given a variable named `x`, if the need arises in the source to assign multiple values to it, in the SSA form, the variable will be represented by different versions like `x1`, `x2`, and so forth. This simplifies the process of tracing a variable's value at any point in the program.

2.5.2.1 Key Characteristics of SSA

- **Unique Assignments:** Every variable in the SSA form gets written to exactly once. This uniqueness in assignment drastically reduces ambiguities during code analysis and optimization.
- **ϕ-functions:** These special functions emerge as a distinct trait of SSA. At control flow merge points, where variables from diverse paths converge, ϕ-functions are employed. For a variable `x` with different values from two different branches, the merge would use a ϕ-function like `x3 = ϕ(x1, x2)`.
- **Clear Data Flow:** With each variable assigned only once, tracking the flow of data becomes notably more streamlined. This property aids in more efficient and direct data flow analysis.

The structure of the SSA, formed by *variables* and their uniquely assigned *values*, combined with the use of *ϕ-functions*, represents programs in a manner that simplifies many aspects of optimization and analysis. Especially, the process of discerning a variable's value and life cycle is made straightforward due to this unique assignment property. As with the three-address code, SSA form serves as a pivotal step in the compilation process, readying the code for subsequent stages of optimization.

3 REFINEMENT TYPES WITH PREDICATES

In order to explore refinement types we will be discussing the works by Jhala and Vazou (2020). This work is a collection of the authors previous efforts in Vazou, Seidel, and Jhala (2014) building the LiquidHaskell extension which opened the way for many implementations of refinement types on top of existing languages. Notable implementation include the work of Vekris, Cosman, and Jhala (2016) adding refinement types for the TypeScript language, the work of Kazerounian et al. (2018) adding refinement types for the Ruby language, and the work of Sammler et al. (2021) integrating refinement types for the C language.

In Jhala and Vazou (2020), the authors elaborate that the type systems of common high level programming languages can allow types to be *refined* with logic predicates. The authors call this technique *Refinement types with predicates* (JHALA; VAZOU, 2020).

Refinement types with predicates allows programmers to constrain existing types by using predicates to assert desired properties of the values they want to describe (JHALA; VAZOU, 2020). They are particularly advantageous because they offer the option to add information to the type system about the invariants and correctness properties a programmer may care about, and, it is done in such a way that, if the programmer desires, no refinement needs to be added and type system can be thought like a typical type system of common higher level languages (JHALA; VAZOU, 2020).

Furthermore, programmers can start with no refinements and incrementally add refinements to ensure important properties about the source program (JHALA; VAZOU, 2020). They could begin with basic safety requirements, e.g., eliminating division by zero and buffer overflow, or guarantee that a function does not receive an empty collection, and then incrementally add to the specification invariants of custom data types (JHALA; VAZOU, 2020). Ultimately going all the way to specifying and verifying the correctness of different procedures at compile-time (JHALA; VAZOU, 2020).

By enabling verification on the same language as the programming language, refinement types bridge implementation and proof together (JHALA; VAZOU, 2020). This approach creates a development cycle were the implementation hits programmers to what properties are important to verify, and the verification hits on how the implementation can be restructured to better express the invariants and enable formal proof (JHALA; VAZOU, 2020).

3.1 EXTENDING THE SIMPLY TYPED LAMBDA CALCULUS WITH REFINEMENTS

Concerning the discussion of the implementation of a type system with refinements with predicates, the work by Jhala and Vazou (2020) explain how to extend the simply typed lambda calculus we explored in Section 2.4 together with the respective inference rules and the newly added typing relations.

3.1.1 Abstract syntax of predicates and constraints

The authors start the construction by defining the abstract syntax for the predicates and constraints syntactical categories as seen in Figure 20. The syntax used by the authors is a taken from the *quatifier-free* fragments of linear arithmetic and uninterpreted functions (JHALA; VAZOU, 2020). The definition of predicates p includes boolean literals, integer literals, variables ranging boolean and integer values, linear arithmetic operations, and boolean operations. Also, the ternary operator

$$\begin{array}{rcll}
\mathsf{p} & ::= & \mathsf{x} \mid \mathsf{y} \mid \mathsf{z} \mid \ldots & \textit{Variables} \\
& \mid & \mathtt{true} \mid \mathtt{false} & \textit{Booleans} \\
& \mid & \mathtt{0} \mid \mathtt{-1} \mid \mathtt{1} \mid \ldots & \textit{Numbers} \\
& \mid & \mathsf{p}_1 \mathbin{\mathtt{+}} \mathsf{p}_2 \mid \mathsf{p}_1 \mathbin{\mathtt{-}} \mathsf{p}_2 \mid \mathsf{p}_1 \mathbin{\mathtt{*}} \mathsf{p}_2 \mid \ldots & \textit{Arithmetic Operations} \\
& \mid & \mathsf{p}_1 \mathbin{\&\&} \mathsf{p}_2 & \textit{Conjunction} \\
& \mid & \mathsf{p}_1 \mathbin{\mathtt{||}} \mathsf{p}_2 & \textit{Disjunction} \\
& \mid & \mathtt{!}\mathsf{p}_1 & \textit{Negation} \\
& \mid & \mathtt{if}\ \mathsf{p}_1\ \mathtt{then}\ \mathsf{p}_2\ \mathtt{else}\ \mathsf{p}_3 & \textit{Conditional} \\
& \mid & \mathsf{p}_1 \bowtie \mathsf{p}_2 & \textit{Interpreted Operations} \\
& \mid & f(\mathsf{p}) & \textit{Uninterpreted Function} \\
\\
\mathsf{c} & ::= & \mathsf{p} & \textit{Predicate} \\
& \mid & \mathsf{c}_1 \wedge \mathsf{c}_2 & \textit{Conjunction} \\
& \mid & \forall \mathsf{x} \mathbin{\mathtt{:}} \mathsf{b}\mathtt{.}\ \mathsf{p} \implies \mathsf{c} & \textit{Implication}
\end{array}$$

Figure 20 – Abstract syntax for refinement predicates and constraints. Adapted from (JHALA; VAZOU, 2020).

$$\mathtt{if}\ \mathsf{p}_1\ \mathtt{then}\ \mathsf{p}_2\ \mathtt{else}\ \mathsf{p}_3$$

was added as an abbreviation to

$$(\mathsf{p}_1 \implies x = \mathsf{p}_2) \wedge (\neg \mathsf{p}_1 \implies x = \mathsf{p}_3)$$

and the operation $\mathsf{p}_1 \bowtie \mathsf{p}_2$ was added to represent all interpreted operators the logic of predicates can be extended with in order to use features from Satisfiability Modulo Theories (SMT) decidable logics (e.g., set operations) (JHALA; VAZOU, 2020). All the other operations are mapped to uninterpreted functions, were the only thing the SMT solver knows about them is the axiom of congruence which says the for all variables x and y, if $x = y$ then $f(x) = f(y)$ (JHALA; VAZOU, 2020). Examples of predicates include the ones we have discussed in the Introduction of this work, we can use `0 <= v` to denote the natural numbers and `0 <= v && v = length(x)` to denote the values of valid indices of an array.

A type checker with refinements produces what is called *Verification Condition* (VC) constraints `c`, which are defined in Figure 20. Constrains are either a single quantifier-free predicate `p`, a conjunction $\mathsf{c}_1 \wedge \mathsf{c}_2$, or an implication of the form $\forall$`x : T.p` $\implies$ `c` (JHALA; VAZOU, 2020). The latter form of a constraint means that for all `x` of a given base type `b`, if `p` holds then so must `c`.

An SMT solver can then verify if a constraint `c` is valid by flattening `c` into a collection of sub-constraints $\mathsf{c}_i = \forall x.\mathsf{p}_i \implies \mathsf{q}_i$ such that $\mathsf{c} \iff \mathsf{c}_i$ (JHALA; VAZOU, 2020). The validity of each c_i is then determined by checking if the quantifier free predicate $\mathsf{p}_i \wedge \neg \mathsf{q}_i$ is not solvable by an SMT solver.

3.1.2 Abstract syntax of terms and types

The abstract syntax for terms and types presented by the authors is summarized in Figure 21.

The terms presented by the authors are similar to the terms explores in Section 2.4. They differ only by the function application, which now must receive a variable as parameter, by the introduction of type annotation terms, and, by the introduction of a recursive binding that allows the use of recursive functions. Also, the constants `c` includes primitives such as integers and primitive functions such as `sum` and `sub` for arithmetic operations. Furthermore, in the definition of types, we find:

- The syntactic variable `b` for the languages primitive types (e.g., the set of all integers `int`);
- The refined types `t` composed of a base type `b` and the definition of a refinement `{x|p}`, which is a pair of a *value variable* `x` and a logical predicate `p` from the SMT logic in Figure 20.
- And, a dependent function type $\mathtt{x}{:}\mathtt{t}_1 \rightarrow \mathtt{t}_2$ composed of a binder `x` of type $\mathtt{t}_1$ which can appear at the refinement predicate of $\mathtt{t}_2$.

In order to reduce verbosity, a refined type which allows all values, written `b{x|true}`, can be abbreviated into its base type `b`.

$$
\begin{array}{rll}
\texttt{e} ::= & \texttt{c} & \textit{Constants} \\
\mid & \texttt{x} & \textit{Variables} \\
\mid & \texttt{if x then e else e} & \textit{If Expression} \\
\mid & \texttt{let x = e in e} & \textit{Let Binding} \\
\mid & \texttt{rec x = e:t in e} & \textit{Recursive Binding} \\
\mid & \lambda\texttt{x.e} & \textit{Function} \\
\mid & \texttt{e x} & \textit{Application} \\
\mid & \texttt{e:t} & \textit{Type Annotation} \\
 & & \\
\texttt{t} ::= & \texttt{b\{x|p\}} & \textit{Refined Base} \\
\mid & \texttt{x:t} \rightarrow \texttt{t} & \textit{Dependent Function} \\
\texttt{b} ::= & \texttt{int} & \textit{Base Integer Type} \\
\mid & \texttt{bool} & \textit{Base Boolean Type} \\
 & & \\
\Gamma ::= & \emptyset & \textit{Empty Context} \\
\mid & \Gamma\texttt{, x : t} & \textit{Variable Binding}
\end{array}
$$

Figure 21 – Abstract syntax for terms, types, and context of a simply typed lambda calculus with refinements. Adapted from (JHALA; VAZOU, 2020).

3.1.3 Variable substitution rules for refinement types

The authors use the notation `t[x := y]` to denote a type substitution where all free occurrences of `y` inside the type `t` are substituted by `x`. For the refined type, the substitution is defined as

$$\texttt{b\{x|p\}[x := y]} = \texttt{b\{x|p\}}$$

for when there is no free occurrences of x in the refined type, and, defined as

$$\texttt{b\{z|p\}[x := y]} = \texttt{b\{z|p[x := y]\}}$$

for when `x` is a free variable in the refinement type. Analogously, the substitution of the refined function type is defined as:

$$
\begin{aligned}
\texttt{(x:t}_1 \rightarrow \texttt{t}_2\texttt{)[x := y]} &= \texttt{x:t}_1\texttt{[x := y]} \rightarrow \texttt{t}_2 \\
\texttt{(z:t}_1 \rightarrow \texttt{t}_2\texttt{)[x := y]} &= \texttt{z:t}_1\texttt{[x := y]} \rightarrow \texttt{t}_2\texttt{[x := y]}
\end{aligned}
$$

3.1.4 Judgments

For statically ensuring that the simply typed lambda calculus with refinement types could be correctly evaluated, Jhala and Vazou (2020) make use of a few new judgments (i.e., relations):

- The *Well-sortedness* judgment $\Gamma \vdash \texttt{p}$ meaning that in the context Γ the predicate $\texttt{p}$ is *well-sorted*, that it, $\texttt{p}$ has boolean type under the context Γ with all refinement types erased using the type relation of the unrefined simply typed lambda calculus (JHALA; VAZOU, 2020).

- The *Well-formedness* judgment $\Gamma \vdash \texttt{t}$ meaning that in the context Γ the type $\texttt{t}$ is *well-formed*, that is, each *t* refinement predicate is boolean-valued under the variables bound in the context or type (JHALA; VAZOU, 2020).

$$\frac{\Gamma, \texttt{x : b} \vdash \texttt{p}}{\Gamma \vdash \texttt{b\{x|p\}}}\ \text{Wf-Base} \qquad\qquad \frac{\Gamma \vdash \texttt{t}_1 \qquad \Gamma, \texttt{x : t}_1 \vdash \texttt{t}_2}{\Gamma \vdash \texttt{x:t}_1 \to \texttt{t}_2}\ \text{Wf-Fun}$$

Figure 22 – Inference rules for the Well-formedness judgment. Adapted from Jhala and Vazou (2020).

The inference rules in Figure 22 define the well-formedness judgment. The rule Wf-Base defines that, for a refinement $\texttt{b\{x|p\}}$ to be well-formed in a context Γ, the predicate $\texttt{p}$ needs to be well-sorted in the context Γ extended with the binding $\texttt{x : b}$ (assuming the erasure of all refinements from Γ) (JHALA; VAZOU, 2020). And, the rule Wf-Fun defines that, for a function type $\texttt{x:t}_1 \to \texttt{t}_2$ to be well-formed, the type $\texttt{t}_1$ needs to be well-formed, and, the type $\texttt{t}_2$ needs to be well-formed with the context extended with the parameter $\texttt{x : t}_1$ (JHALA; VAZOU, 2020).

- The *Entailment* judgment $\Gamma \vdash \texttt{c}$ meaning that in the context Γ the constraint $\texttt{c}$ is *valid*, that is, the constraint $\texttt{c}$ 'is true' (JHALA; VAZOU, 2020).

$$\frac{\texttt{SmtValid(c)}}{\emptyset \vdash \texttt{c}}\ \text{Ent-Empty} \qquad\qquad \frac{\Gamma \vdash \forall \texttt{x : b. p} \implies \texttt{c}}{\Gamma, \texttt{x : b\{x|p\}} \vdash \texttt{c}}\ \text{Ent-Reduce}$$

Figure 23 – Inference rules for the entailment judgment. Adapted from Jhala and Vazou (2020).

The inference rules in Figure 23 define the entailment judgment. The rule Ent-Empty defines that the context Γ entails the constraint $\texttt{c}$ given that $\texttt{c}$ is SMT solvable (JHALA; VAZOU, 2020). Furthermore, the rule Ent-Reduce defines how to reduce the context into the verification condition by removing bindings from the context and translating them into constraints (JHALA; VAZOU, 2020).

- The *Subtyping* judgment $\Gamma \vdash \texttt{t}_1 \prec: \texttt{t}_2$ meaning that in the context Γ the type $\texttt{t}_1$ is a subtype of $\texttt{t}_2$, that is, all values of $\texttt{t}_1$ are present in $\texttt{t}_2$ but the contrary may not be true (JHALA; VAZOU, 2020).

$$\frac{\Gamma \vdash \forall \mathtt{x}_1 \mathtt{\ :\ b.\ } \mathtt{p}_1 \Longrightarrow \mathtt{p}_2\mathtt{[x}_2\mathtt{\ :=\ x}_1\mathtt{]}}{\Gamma \vdash \mathtt{b\{x}_1\mathtt{|p}_1\mathtt{\}} \prec: \mathtt{b\{x}_2\mathtt{|p}_2\mathtt{\}}}\ \text{Sub-Base}$$

$$\frac{\Gamma \vdash \mathtt{t}_{2.1} \prec: \mathtt{t}_{1.1} \qquad \Gamma\mathtt{,\ x}_2 \mathtt{\ :\ t}_{2.1} \vdash \mathtt{t}_{1.2} \prec: \mathtt{t}_{2.2}}{\Gamma \vdash \mathtt{x}_1\mathtt{:t}_{1.1} \rightarrow \mathtt{t}_{1.2} \prec: \mathtt{x}_2\mathtt{:t}_{2.1} \rightarrow \mathtt{t}_{2.2}}\ \text{Sub-Fun}$$

Figure 24 – Inference rules for the Subtyping judgment. Adapted from Jhala and Vazou (2020).

The inference rules in Figure 24 define the subtyping judgment. The rule Sub-Base defines that, for a refined type $\mathtt{b\{x}_1\mathtt{|p}_1\mathtt{\}}$ to be a subtype of $\mathtt{b\{x}_2\mathtt{|p}_2\mathtt{\}}$, the context has to entail the constraint $\forall \mathtt{x}_1 \mathtt{\ :\ b.\ } \mathtt{p}_1 \Longrightarrow \mathtt{p}_2\mathtt{[x}_2\mathtt{\ :=\ x}_1\mathtt{]}$ which means that if $\mathtt{p}_1$ holds then so must $\mathtt{p}_2$ with all free occurrences of $\mathtt{x}_2$ substituted by $\mathtt{x}_1$ (JHALA; VAZOU, 2020). And, the rule Sub-Fun decomposes the subtype of function types into the contravariant subtype of the input types and the covariant subtype of the return types (JHALA; VAZOU, 2020).

- The *Synthesis* judgment $\Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}$ meaning that in the context Γ the type $\mathtt{t}$ can be generated, or synthesized, for the term $\mathtt{e}$ (JHALA; VAZOU, 2020). This judgment is relative to the typing relation for the unrefined simply typed lambda calculus in Section 2.4.

  The inference rules in Figure 25 define the synthesis judgment. They work together with the *checking* judgment inference rules to build the *bidirectional typing* system described in Section 3.1.5.

- And, the *Checking* judgment $\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{t}$ meaning that in the context Γ the type $\mathtt{t}$ is a valid type for the term $\mathtt{e}$, that is, constraining the term $\mathtt{e}$ to *synthesize* a type $\mathtt{s}$ where $\mathtt{s} \prec: \mathtt{t}$ (JHALA; VAZOU, 2020). This judgment is used to verify that a term is of given annotated type and to push the type annotation inside it's sub-terms to get localized type obligations for inner expressions (JHALA; VAZOU, 2020).

  The inference rules in Figure 26 define the checking judgment, complementing the inference rules for the synthesis judgment.

3.1.5 Bidirectional typing

The typing relation $\Gamma \vdash \mathtt{x\ :\ T}$ presented in Section 2.4.2 for the simply typed lambda calculus proves that the program is sound by synthesizing a type for each leaf term on the abstract tree and then subsequently synthesizing the types for the intermediate terms until we synthesize the type for the root term thus proving the program's soundness traversing the abstract tree in a single direction, i.e., from the leaves to the root.

The bidirectional typing rules proposed by Jhala and Vazou (2020) levers the type synthesis of terms by combining the synthesis with checking judgments. We will explore the inference rules present in Figures 25 and 26 by exploring how they prove the types for each term `e` described in Figure 21.

$$\frac{\mathtt{x : t} \in \Gamma \qquad \mathtt{self(x, t)} = \mathtt{s}}{\Gamma \vdash \mathtt{x} \Rightarrow \mathtt{s}}\ \text{Syn-Var} \qquad\qquad \frac{\mathtt{prim(c)} = \mathtt{t}}{\Gamma \vdash \mathtt{c} \Rightarrow \mathtt{t}}\ \text{Syn-Const}$$

$$\frac{\Gamma \vdash \mathtt{e} \Rightarrow \mathtt{y{:}t_1} \rightarrow \mathtt{t_2} \qquad \Gamma \vdash \mathtt{x} \Leftarrow \mathtt{t_1}}{\Gamma \vdash \mathtt{e\ x} \Rightarrow \mathtt{t_2[y := x]}}\ \text{Syn-App} \qquad \frac{\Gamma \vdash \mathtt{t} \qquad \Gamma \vdash \mathtt{e} \Leftarrow \mathtt{t}}{\Gamma \vdash \mathtt{e{:}t} \Rightarrow \mathtt{t}}\ \text{Syn-Ann}$$

Figure 25 – Inference rules for the Synthesis judgment. Adapted from Jhala and Vazou (2020).

$$\frac{\Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t_1} \qquad \Gamma \vdash \mathtt{t_1} \prec: \mathtt{t_2}}{\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{t_2}}\ \text{Chk-Syn} \qquad\qquad \frac{\Gamma, \mathtt{x : t_1} \vdash \mathtt{e} \Leftarrow \mathtt{t_2}}{\Gamma \vdash \lambda\mathtt{x.e} \Leftarrow \mathtt{x{:}t_1} \rightarrow \mathtt{t_2}}\ \text{Chk-Fun}$$

$$\frac{\Gamma \vdash \mathtt{e_1} \Rightarrow \mathtt{t_1} \qquad \Gamma, \mathtt{x : t_1} \vdash \mathtt{e_2} \Leftarrow \mathtt{t_2}}{\Gamma \vdash \mathtt{let\ x = e_1\ in\ e_2} \Leftarrow \mathtt{t_2}}\ \text{Chk-Let}$$

$$\frac{\Gamma \vdash \mathtt{t_1} \qquad \Gamma, \mathtt{x : t_1} \vdash \mathtt{e_1} \Leftarrow \mathtt{t_1} \qquad \Gamma, \mathtt{x : t_1} \vdash \mathtt{e_2} \Leftarrow \mathtt{t_2}}{\Gamma \vdash \mathtt{rec\ x = e_1{:}t_1\ in\ e_2} \Leftarrow \mathtt{t_2}}\ \text{Chk-Rec}$$

$$\frac{\begin{array}{c}\mathtt{y} \notin \Gamma \qquad \Gamma \vdash \mathtt{x} \Leftarrow \mathtt{bool} \\ \Gamma, \mathtt{y : int\{y|x\}} \vdash \mathtt{e_1} \Leftarrow \mathtt{t} \qquad \Gamma, \mathtt{y : int\{y|!x\}} \vdash \mathtt{e_2} \Leftarrow \mathtt{t}\end{array}}{\Gamma \vdash \mathtt{if\ x\ then\ e_1\ else\ e_2} \Leftarrow \mathtt{t}}\ \text{Chk-If}$$

Figure 26 – Inference rules for the Checking judgment. Adapted from Jhala and Vazou (2020).

The terms that can be synthesized and their respective inference rules for their synthesis definition are:

- Constant terms `c` synthesize their primitive type denoted by `prim(c)` as formalized by the rule Syn-Const. For example, the literals `0` and `1` for the unrefined type `int` are mapped to singleton types as in

$$\begin{aligned}\texttt{prim(0)} &= \texttt{int\{x|x == 0\}}\\ \texttt{prim(1)} &= \texttt{int\{x|x == 1\}}\\ \texttt{prim(true)} &= \texttt{bool\{x|x\}}\\ \texttt{prim(false)} &= \texttt{bool\{x|!x\}}\end{aligned}$$

Also, primitive functions as arithmetic operations are assigned to types that reflect their semantics as in

$$\begin{aligned}\texttt{prim(add)} &= \texttt{x:int} \to \texttt{y:int} \to \texttt{int\{z|z == x + y\}}\\ \texttt{prim(sub)} &= \texttt{x:int} \to \texttt{y:int} \to \texttt{int\{z|z == x - y\}}\end{aligned}$$

and comparison operations as in

$$\begin{aligned}\texttt{prim(leq)} &= \texttt{x:int} \to \texttt{y:int} \to \texttt{bool\{z|z == x < y\}}\\ \texttt{prim(get)} &= \texttt{x:int} \to \texttt{y:int} \to \texttt{bool\{z|z == x >= y\}}\end{aligned}$$

- Variable terms `x` synthesize the type resulting from `self(x, t)` if `x : t` $\in \Gamma$ through the use of the rule Syn-Var where the function `self` is defined as

  $$\texttt{self(x, t)} = \begin{cases} \texttt{b\{y| p \&\& y == x\}} & \text{if } \texttt{t} = \texttt{b\{y|p\}} \\ \texttt{t} & \text{otherwise} \end{cases}$$

  The use of `self` guaranties that we further strengthen base type refinements by bring the value of `x` into the refinement predicate of `t`. For example, consider the function `abs` for calculating the absolute value on an integer `x`

  $$\texttt{abs} = \lambda\texttt{x. let c = leq(0, x) in if (c) then (x) else (sub 0 x)}$$

  of type

  $$\texttt{abs : x:int} \to \texttt{int\{y| y >= 0 \&\& y >= x\}}$$

  If the variable term `x` had synthesized type `int{x|true}`, the body term of the `abs` function would never be a subtype of its output type, while, when extending the type predicate into the type `int{y| true && x == y}` the generated validity constraint can reason on the value of `x` and check the validity of the function `abs` (JHALA; VAZOU, 2020).

- Application terms `e x` synthesize the output type of the function type synthesized for the term `e` with its input binder substituted by the real argument variable `x` through rule Syn-App. Also, the variable `x` is constrained by the input type by the checking judgment $\Gamma \vdash \texttt{x} \Leftarrow \texttt{t}_1$ in the premises of Syn-App. For example, given the context

  $$\Gamma = \emptyset, \texttt{nat : int\{x|x > 0\}}, \texttt{one : int\{x| x == 1\}}$$

  the term `add nat one` would synthesize

  $$\Gamma \vdash \texttt{add nat one} \Rightarrow \texttt{int\{z|z == nat + one\}}$$

  The only inference rule from the checking judgment definition that can check the premises

  $$\Gamma \vdash \texttt{nat} \Leftarrow \texttt{int} \quad \text{and} \quad \Gamma \vdash \texttt{one} \Leftarrow \texttt{int}$$

  is the rule Chk-Syn, which, will make sure that the variables `nat` and `one` synthesize types that are subtypes of the respective arguments type for the function `add`.

  Furthermore, Jhala and Vazou (2020) explain that the function application must receive variables as its arguments in order to properly substitute the input binders in the function's output type. The authors explain how to expand the type syntax to include existential types ∃`x:s.t` in order to bypass this constraint but, in order to ease exposition and implementation, Jhala and Vazou (2020) opt by keeping the constraint because a program can be easily modified to follow the constraint during analysis. For example, the program

  ```
  add (add 5 5) one
  ```

  can be easily translated into

  ```
  let aux = add 5 5 in add aux one
  ```

  instead of further complicating the set of inference rules for allowing non-variable terms as arguments.

- Annotation terms `e:t` synthesize its annotated type `t` if the annotated term `e` can be checked against the type `t` and the type `t` is well-formed as defined by rule Syn-Ann.

In order to complete the discussion of the bidirectional typing rules, we discuss the terms that can be checked and their checking definition rules:

- Function terms `λx.e` do not directly synthesize a type, those can only be checked against a type through the Chk-Fun inference rule. A function can be checked against the type $\texttt{x:t}_1 \rightarrow \texttt{t}_2$ if its body `e` can be checked against $\texttt{t}_2$ with the context extended with the argument binding $\texttt{x : t}_1$ as defined by the rule Chk-Fun.

- Let Binding terms $\texttt{let x = e}_1 \texttt{ in e}_2$ also do not directly synthesize a type requiring to be checked through the Chk-Let inference rule. A let binding can be checked against the type $\texttt{t}_2$ if $\texttt{e}_2$ can be checked against $\texttt{t}_2$ with the context extended with the binding $\texttt{x : t}_1$ which has to be synthesized for $\texttt{e}_1$.

- Recursive Binding terms $\texttt{rec x = e}_1\texttt{:t}_1 \texttt{ in e}_2$ is a similar case to let bindings and can be checked through the Chk-Rec inference rule. A recursive binding differs from the let binding by requiring a type annotation in the expression $\texttt{e}_1$, and by pushing the type binding $\texttt{x : t}_1$ into the checking of $\texttt{e}_1$ instead of synthesizing its type.

- If Expression terms $\texttt{if x then e}_1 \texttt{ else e}_2$ also do not directly synthesize a type requiring to be checked through the Chk-If inference rule. An if expression can be checked to have type `t` in a context Γ if both $\texttt{e}_1$ and $\texttt{e}_2$ can be checked to have type `t`. Furthermore, the inference rule Chk-If differs from a classic definition of an if expression term by adding into the context a *fresh* `y` variable (i.e., not present in the context) bound to a refinement that captures the exact value of the condition `x` when checking the terms $\texttt{e}_1$ and $\texttt{e}_2$ (JHALA; VAZOU, 2020). If this binding was not present in the context, the checking relation

$$\texttt{not} = \lambda\texttt{x.if x then false else true}$$
$$\Gamma \vdash \texttt{not} \Leftarrow \texttt{x:bool} \rightarrow \texttt{bool\{b|b == !x\}}$$

  would not be able to prove that the output type of the `not` function is the inverse of the condition value receive as input (JHALA; VAZOU, 2020).

Last, the terms for constants, variables, applications, and, annotations, can be checked by the use of the Chk-Syn inference rule. The subsumpiton rule Chk-Syn connects the checking and synthesis judgments by defining that if a term `e` synthesized a type $\texttt{t}_1$ and the type $\texttt{t}_1$ is subsumed by type $\texttt{t}_2$ (i.e., $\texttt{t}_1$ is subtype of $\texttt{t}_2$) then we can check `e` against $\texttt{t}_2$.

3.1.6 An implementation of a verification condition generator

In order to verify the soundness of a program for the proposed simply typed lambda calculus with refinements, Jhala and Vazou (2020) describes an implementation of a verification condition (VC) generator. The proposed generator takes as input the program's abstract syntax and outputs a VC constraint `c` which can be validated by an SMT solver and implies the program's soundness (JHALA; VAZOU, 2020). More specifically, the authors describe an algorithm implementation for the subtyping, synthesis and checking judgments.

The algorithms make use of an implication constraint written $(\mathtt{x} :: \mathtt{t}) \implies \mathtt{c}$ defined as

$$(\mathtt{x} :: \mathtt{t}) \implies \mathtt{c} = \begin{cases} \forall \mathtt{x{:}b.\ p[y := x]} \implies \mathtt{c} & \text{if } \mathtt{t} = \mathtt{b\{y|p\}} \\ \mathtt{c} & \text{otherwise} \end{cases}$$

The subtyping relation can be implemented as a function `sub` that takes two types $\mathtt{t}_1$ and $\mathtt{t}_2$ as input and outputs a constraint `c` which the validity of `c` implies the subtyping relation $\mathtt{t}_1 \prec: \mathtt{t}_2$ formalized by Jhala and Vazou (2020) as the following proposition:

$$\text{if } \mathtt{sub(t_1,\ t_2)} = \mathtt{c} \text{ and } \Gamma \vdash \mathtt{c} \text{ then } \Gamma \vdash \mathtt{t}_1 \prec: \mathtt{t}_2$$

The function `sub` is implemented as shown in Figure 27 where each definition is relative to the inference rules Sub-Base and Sub-Fun from Figure 24.

$$\begin{aligned} \mathtt{sub(b\{x_1,\ p_1\},\ b\{x_2,\ p_2\})} &= \forall \mathtt{x{:}b.\ p_1} \implies \mathtt{p_2[x_2 := x_1]} \\ \mathtt{sub(x_1{:}t_{1.1} \rightarrow t_{1.2},\ x_2{:}t_{2.1} \rightarrow t_{2.2})} &= c_1 \wedge (\mathtt{x_2} :: \mathtt{t_{2.1}}) \implies c_2 \\ &\quad \text{where:} \\ &\quad\quad c_1 = \mathtt{sub(t_{1.1},\ t_{2.1})} \\ &\quad\quad c_2 = \mathtt{sub(t_{1.2}[x_1 := x_2],\ t_{2.2})} \end{aligned}$$

Figure 27 – Subtyping function for the subtyping relation. Adapted from Jhala and Vazou (2020).

The synthesis relation can be implemented as a function `synth` that takes as input the context Γ and a term `e` and outputs the tuple `(c, t)` where `t` is the type of `e` in Γ and the constraint `c`'s validity implies the synthesis relation as formalized by Jhala and Vazou (2020) in the proposition:

$$\text{if } \mathtt{synth(}\Gamma\mathtt{,\ e)} = \mathtt{(c,\ t)} \text{ and } \Gamma \vdash \mathtt{c} \text{ then } \Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}$$

The function `synth` is implemented as shown in Figure 28 where each definition is relative to the inference rules Syn-Var, Syn-Const, Syn-Ann, and Syn-App from Figure 25.

Last, the checking relation can be implemented as a function `check` that takes as input the context Γ, a term `e`, and, an expected type `t`, and output a constraint `c` which validity implies the synthesis relation as formalized by Jhala and Vazou (2020) in the proposition:

$$\text{if } \mathtt{check(}\Gamma\mathtt{,\ e,\ t)} = \mathtt{c} \text{ and } \Gamma \vdash \mathtt{c} \text{ then } \Gamma \vdash \mathtt{e} \Leftarrow \mathtt{t}$$

$$\begin{aligned}
\texttt{synth}(\Gamma,\ \texttt{x}) &= (\texttt{true},\ \texttt{self}(\texttt{x},\ \texttt{t}))\\
&\quad \text{where:}\\
&\qquad \texttt{x : t} \in \Gamma\\
\texttt{synth}(\Gamma,\ \texttt{c}) &= (\texttt{true},\ \texttt{prim}(\texttt{c}))\\
\texttt{synth}(\Gamma,\ \texttt{e y}) &= (c_1 \wedge c_2,\ \texttt{t}_2[\texttt{x := y}])\\
&\quad \text{where:}\\
&\qquad (c_1,\ \texttt{x:t}_1 \rightarrow \texttt{t}_2) = \texttt{synth}(\Gamma,\ \texttt{e})\\
&\qquad c_2 = \texttt{check}(\Gamma,\ \texttt{y},\ \texttt{t}_1)\\
\texttt{synth}(\Gamma,\ \texttt{e:t}) &= (c,\ \texttt{t})\\
&\quad \text{where:}\\
&\qquad c = \texttt{check}(\Gamma,\ \texttt{e},\ \texttt{t})
\end{aligned}$$

Figure 28 – Synthesis function for the synthesis relation. Adapted from Jhala and Vazou (2020).

The function `check` is implemented as shown in Figure 29 where each definition is relative to the inference rules Chk-Fun, Chk-Let, Chk-Rec, Chk-If, and Chk-Syn, present in Figure 26.

$$\begin{aligned}
\texttt{check}(\Gamma,\ \texttt{e},\ \texttt{t}_2) &= c_1 \wedge c_2\\
&\text{where:}\\
&\quad (c_1,\ \texttt{t}_1) = \texttt{synth}(\Gamma,\ \texttt{e})\\
&\quad c_2 = \texttt{sub}(\texttt{t}_1,\ \texttt{t}_2)\\
\texttt{check}(\Gamma,\ \lambda\texttt{x.e},\ \texttt{x:t}_1 \rightarrow \texttt{t}_2) &= (\texttt{x} :: \texttt{t}_1) \implies c\\
&\text{where:}\\
&\quad c = \texttt{check}(\Gamma_1,\ \texttt{e},\ \texttt{t})\\
&\quad \Gamma_1 = \Gamma,\ \texttt{x : t}_1\\
\texttt{check}(\Gamma,\ \texttt{let x = e}_1\ \texttt{in e}_2,\ \texttt{t}_2) &= c_1 \wedge (\texttt{x} :: \texttt{t}_1) \implies c_2\\
&\text{where:}\\
&\quad (c_1,\ \texttt{t}_1) = \texttt{synth}(\Gamma,\ \texttt{e}_1)\\
&\quad c_2 = \texttt{check}(\Gamma_1,\ \texttt{e}_2,\ \texttt{t}_2)\\
&\quad \Gamma_1 = \Gamma,\ \texttt{x : t}_1\\
\texttt{check}(\Gamma,\ \texttt{rec x = e}_1\texttt{:t}_1\ \texttt{in e}_2,\ \texttt{t}_2) &= c_1 \wedge c_2\\
&\text{where:}\\
&\quad c_1 = \texttt{check}(\Gamma_1,\ \texttt{e}_1,\ \texttt{t}_1)\\
&\quad c_2 = \texttt{check}(\Gamma_1,\ \texttt{e}_2,\ \texttt{t}_2)\\
&\quad \Gamma_1 = \Gamma,\ \texttt{x : t}_1\\
\texttt{check}(\Gamma,\ \texttt{if x then e}_1\ \texttt{else e}_2,\ \texttt{t}) &= c_1 \wedge c_2\\
&\text{where:}\\
&\quad c_1 = (\texttt{y} :: \texttt{int\{x|x\}}) \implies \texttt{check}(\Gamma,\ \texttt{e}_1,\ \texttt{t})\\
&\quad c_2 = (\texttt{y} :: \texttt{int\{x|!x\}}) \implies \texttt{check}(\Gamma,\ \texttt{e}_2,\ \texttt{t})
\end{aligned}$$

Figure 29 – Synthesis function for the synthesis relation. Adapted from Jhala and Vazou (2020).

4 THE LLVM INTERMEDIATE REPRESENTATION

LLVM, an acronym for Low-Level Virtual Machine, is a pioneering compiler infrastructure renowned for its versatility and reusability. LLVM supplies a suite of reusable modules crucial for constructing compilers. Distinctively, it's designed with language agnosticism in mind, making it a premier choice for compiling a broad spectrum of programming languages, including but not limited to C, C++, Rust, and Swift (DENISOV; PANKEVICH, 2018).

4.1 LLVM'S INTERMEDIATE REPRESENTATION (IR)

At the heart of LLVM's design is its Intermediate Representation (IR), often referred to as LLVM IR. This IR serves as a three-address code, meticulously designed to delineate programs in a format that is both low-level and independent of any particular platform (LEE et al., 2018).

4.1.1 Characteristics of LLVM IR

- **SSA-based Representation:** LLVM IR employs a Static Single Assignment (SSA) form. A fundamental characteristic of SSA is its insistence that each variable be assigned precisely once, making it a prime structure for optimizing and analyzing code efficiently.
- **Typed Language:** Another striking aspect of LLVM IR is its strong typing system. Analogous to those in many high-level programming languages, types in LLVM IR range from integers and floating-point numbers to more complex structures like pointers.
- **Structure of LLVM IR:** Programs in LLVM IR are intricately constructed using a mix of instructions and basic blocks. While instructions are the workhorses performing operations on values, basic blocks present these instructions in sequences ensuring sequential execution. Furthermore, control flow, an essential aspect of any programming representation, finds its expression in LLVM IR through conditional branches and unequivocal jumps.

4.1.2 Optimizations in LLVM

One of the crowning features of LLVM IR is its powerful optimization capabilities. The LLVM framework equips developers with a battery of optimization passes, tailored to refine the IR and enhance the efficiency of the resultant code (LEE et al., 2018). These optimization techniques span across various strategies, such as: Common subexpression elimination; Dead code elimination; Loop optimizations.

The LLVM optimizer, with its robust analytical prowess, examines code to judiciously apply these optimizations. The optimizer's decisions are rooted in the inherent properties of the IR, notably the SSA form and the embedded type information.

5 THE EKITAI LANGUAGE

In this chapter we will describe the design of the *Ekitai* programming and the implementation of its front end to the LLVM intermediate representation.

5.1 THE EKITAI'S LEXER

As explored in section Section 2.2 we define the lexical aspects of the Ekitai programming language presented in Figures 30 and 31.

We decided to use an automatic lexer generator since we found no advantage in building one by hand and the automatic parser generator used was capable of lexing all token patterns in the specification.[1]

`digit` → [`0-9`]
`alpha` → [`a-zA-Z`]
`word` → [`[:digit:][:alpha:]_`]

Figure 30 – Building block regular definitions for token patterns of Figure 31.

`Comma` → `,`
`Collon` → `:`
`SemiColon` → `;`
`OpenParenthesis` → `(`
`CloseParenthesis` → `)`
`OpenBraces` → `{`
`CloseBraces` → `}`
`Equals` → `=`
`Plus` → `+`
`Minus` → `-`
`Asterisk` → `*`
`Ampersand` → `&`

`Slash` → `/`
`Percent` → `%`
`Greater` → `>`
`Less` → `<`
`Exclamation` → `!`
`Pipe` → `|`
`DoubleCollon` → `::`
`DoubleEquals` → `==`
`ExclamationEquals` → `!=`
`GreaterEquals` → `>=`
`LessEquals` → `<=`
`DoublePipe` → `||`
`DoubleAmpersand` → `&&`

`ThinArow` → `->`
`FatArow` → `=>`
`FnKw` → `fn`
`LetKw` → `let`
`IfKw` → `if`
`ElseKw` → `else`
`TrueKw` → `true`
`FalseKw` → `false`
`TypeKw` → `type`
`MatchKw` → `match`
`NewKw` → `new`

`Identifier` → [`[:alpha:]_`][`[:word:]`]*
`Integer` → `[:digit:]`[`[:digit:]_`]*[`[:Identifier:]`ε]

Figure 31 – Token names and patterns for the Ekitai lexer.

Note in Figure 31 that the token name `Identifier` has, in its described language, common strings with token names for keywords, i.e., token names ending in `Kw`.

---

[1] The automatic lexer generator used in this work is Logos (HIRSZ, 2020).

The automatic lexer algorithm solves this problem by applying two disambiguation rules:

- Longer matching lexemes have priority over shorter matching lexemes, i.e., `>=` has higher priority than `>` for token names `GreaterEquals` and `Greater`.
- Groups have lower priority that singular regular expressions, i.e., [`ab`] has less priority than `a` and `b`, and, permutations have the lowest priority, i.e, a^*.

Furthermore, the token name `Integer` must start with a digit and is followed by any number of digits and underscores, allowing to separate long numbers (i.e., `100_000`), with an optional identifier at the end, allowing for the use of suffixes (i.e., `42i64`).

5.2 THE EKITAI'S PARSER

The Ekitai's grammar was built with a handwritten Pratt parser in mind (PRATT, 1973). As explored in section Section 2.3 we will be presenting Ekitais syntax. In order to keep the grammar out of clutter, we will present the grammar using simplified token names. Instead of writing `LetKw` or `DoubleAmpersand` we will stick to writing `let` and `&&` respectively. Furthermore, we will use `Id` for short of `Identifier` and `Int` for short of `Integer`.

First we define the syntax structure for the source program's top level items by presenting the grammar productions in Figure 32.

```
   SourceFile ::=  ItemList
     ItemList ::=  Item ItemList | ε
         Item ::=  FnDef | TypeDef
      TypeDef ::=  'type' Name '{' ValueConsList '}'
ValueConsList ::=  ValueCons ',' ValueConsList
                |  ValueCons | ε
    ValueCons ::=  Name '(' TypeList ')'
     TypeList ::=  Type ',' TypeList
                |  Type | ε
        FnDef ::=  'fn' Name '(' ParamList ')' '->' BlockExpr
    ParamList ::=  Param ',' ParamList
                |  Param | ε
        Param ::=  Name ':' Type
         Name ::=  'Id'
```

Figure 32 – Top level grammar productions for the Ekitai's parser grammar.

The variable `SourceFile` is the start symbol of the Ekytai's syntax grammar. It generates a list of Items `FnDef` and `TypeDef` for function definitions and type definitions respectively.

From the grammar in Figure 32, we are able to see the structure of the top level constructs. For example, the following program illustrates a `SourceFile` with two `Item` in the `ItemList`:

```
type SomeTypeName {
  SomeConstructorName(Type1, Type2, ... ),
  ...
}
fn some_fn_name(arg_name1: Type1, arg_name_2: Type2, ...) -> {
  ...
}
```

When translating the Ekitai's parse tree to Ekitai's we use the `Path` variable to index the constructors inside the `type` definition and functions names inside modules, for example the path

`SomeTypeName::SomeConstructorName`

will index the constructor

`SomeConstructorName`

inside the type definition for

`SomeTypeName`

and respectively for functions of different source files. The `Path` variable is defined on Figure 33 as part of Ekitai's grammar for terms and is used to uniquely identify functions and types of the Ekitai language.

Beyond the top level items, we still have to define the variables for `Type` and for `BlockExpr`. The `BlockExpr` variable is part of the grammar for Ekitai's terms as shown in Figure 33. While the `Type` variable is part of the grammar for Ekitai's types defined in Figure 34.

The syntax for refinements defined in Figure 34 allows for any expression as the refinement predicate. We do that in order to reuse the parser for `Expr` and leave to the semantic analysis to restrict what sentences from `Expr` are allowed as refinement predicates.

The techniques used by Ekitai's Pratt parser were explored in Section 2.3. If the reader wants to know the specifics of the implementation we guide to source files of the modules `syntax` and `parser` of Ekitai's source code at Appendix A.

After construction of the CST datastructure containg the parse tree, Ekitai's front-end translates its CST to an intermediate AST. The intermediate AST is derived of the lambda calculus with refinements presented in Chapter 3. The techniques employed in this translation were explored in Section 2.3.4. If the reader wishes

```
Expr ::= Literal
       | Path
       | '(' Expr ')'
       | Expr InfixOp Expr
       | PrefixOp Expr
       | Expr '(' ArgList ')'
       | BlockExpr
       | 'if' Expr BlockExpr
         'else' BlockExpr
       | MatchExpr
       | 'new' Expr

      InfixOp ::= '+' | '-' | '*' | '/' | '%'
                | '>' | '>=' | '<' | '<='
                | '==' | '!=' | '&&' | '||'
     PrefixOp ::= '-' | '!' | '*' | '&'
      ArgList ::= Expr ',' ArgList
                | Expr | ε
    BlockExpr ::= '{' StatementList Expr '}'
StatementList ::= Statement ';' StatementList
                | ε
    MatchExpr ::= 'match' Expr '{' CaseList '}'
     CaseList ::= Pattern '=>' Expr ',' CaseList
                | ε

Statement ::= 'let' Name '=' Expr
  Pattern ::= Path '(' NameList ')'
 NameList ::= Name ',' NameList
            | Name | ε
     Path ::= Name '::' Name
            | Name
  Literal ::= 'Int' | 'true' | 'false'
```

Figure 33 – Expression grammar productions for the Ekitai's parser grammar.

```
Type ::= Name
       | '*' Type
       | '{' Name ':' Type '|' Expr '}'
```

Figure 34 – Type grammar productions for the Ekitai's parser grammar.

to explore the details of the AST implementation we guide to source files of the module `hir`, submodule `semantic_ir`, of Ekitai's source code at Appendix A which contains all structs for dealing with Ekitai's higher level intermediate representation. We will be defining Ekitai's AST formally and exploring it's type checking algorithms in Section 5.3.

5.3 THE EKITAI'S TYPE SYSTEM

Ekitai's type-checker aimins to provide developers with a powerful yet intuitive type system by enabling the use of predicates to prove properties on terms of the language. Drawing from the theoretical foundation detailed in Chapter 3, this type system is an implementation of refinement types for a subset of lambda calculus.

5.3.1 Abstract Syntax Tree (AST)

Central to understanding Ekitai's type system is its AST (see Figure 35). The Ekitai's AST is a derivation of the abstract syntax presented back in Section 3.1, which significantly eases the process of type-checking since we can use the same inference rules described to type-check programs that where written in Ekitai's Syntax.

```
    SourceFile ::= ModItem SourceFile | ε
       ModItem ::= FnDef | TyDef
         TyDef ::= type Name = ValueConsList
 ValueConsList ::= ValueCons | ValueConsList | ε
     ValueCons ::= Name TypeList
      TypeList ::= Type TypeList | ε
         FnDef ::= fn Name: FnType = Term
          Name ::= id
          Path ::= Name Path | ε

          Term ::= let Name = Term in Term
                 | if Name then Term else Term
                 | Int | Bool | Name
                 | Term x

          Type ::= RefType | FnType
       RefType ::= {Name: BaseType | Term}
        FnType ::= Name:RefType -> Type
      BaseType ::= I32 | I64 | Bool

             Γ ::= ∅
                 | Γ, Path : Type
```

Figure 35 – Ekitai's AST derived from the lambda calculus with refinements from Section 3.1.

We employed two derivations in Ekitai's AST compared to the lambda calculus with refinements detailed in Section 3.1:

- First, we added toplevel items for type and funciton definitions;
- Seccond, we removed the function term of the language.

The first one enables us to bring everything into context before typechecking the terms inside functions. While the seccond one was employed to simplify the subsequent translation into LLVM-IR by avoiding the need to create unamed closure objects and deal with capturing variables from the function's context.

Next we will explore some key aspects of Ekitai's AST and typechecker implementations. Examples where taken from module `hir`, submodules `liquid` and `check`, found in Appendix A

```
// Base type with a refinement predicate: b{x|p}
struct RefinedBase {
    pub base: Type,
    pub binder: Name,
    pub predicate: Predicate,
}

// x:t -> t
struct DependentFunction {
    pub parameter: (Name, RefinedBase),
    pub tail_type: Box<RefinedType>,
}

// t
enum RefinedType {
    Base(RefinedBase),
    Fn(DependentFunction),
}

pub enum Type {
    AbstractDataType(TypeDefinitionId /* internal ID for our AST database */),
    FunctionDefinition(CallableDefinitionId /* internal ID for our AST database */),
    Pointer(Box<Type>),
    Scalar(ScalarType),
}

pub enum ScalarType {
    Integer(IntegerKind /* I32, I64 */),
    Boolean,
}

struct Context {
    bindings: Vec<(Path, RefinedType)>,
}
```

In this architecture:

- The `RefinedBase` is a structure that encapsulates the base type along with a refinement predicate.

- The `DependentFunction` represents functions that possess a dependency on a specific parameter type.
- `RefinedType` serves as an overarching enum, capturing the essence of both basic and functional types.
- The differentiation between abstract data types, function definitions, pointers, and scalars is captured by the `Type` enum.

5.3.2 Predicates and Constraints

Predicates and constraints form the bedrock of Ekitai's type-checking. They define the conditions under which types are valid, and how they interact with one another.

```rust
pub enum Predicate {
  Variable(Name),
  Boolean(bool),
  Integer(u128),
  // contains arithmetic and boolean operators
  Binary(BinaryOperator, Box<Predicate>, Box<Predicate>),
  Unary(UnaryOperator, Box<Predicate>),
}

enum Constraint {
  Predicate(Predicate),
  Implication {
    binder: Name,
    base: Type,
    antecedent: Predicate,
    consequent: Box<Self>,
  },
  Conjunction(Box<Self>, Box<Self>),
}
```

The given code follows from Section 3.1.1, `Predicate` encompasses several fundamental constructs, like variables, booleans, and integers. It also includes unary and binary operations, enabling more complex type validations. On the other hand, `Constraint` brings together predicates, implications, and conjunctions, offering a robust mechanism to express and check type requirements.

5.3.3 Type Inference and Checking

Ekitai employs advanced type inference algorithms to predict the type of a given term. This prediction, coupled with constraints, becomes instrumental in type validation. The following code follows from subsection 3.1.5:

```
// returns the type of the term and a constraint that must be satisfied
fn synth_type(self, term_id: TermId) -> (Self, RefinedType, Option<Constraint>)

// checks that the type of the lesser term is a subtype of the greater type
fn subtype(self, lesser: RefinedBase, greater: RefinedBase) -> (Self, Constraint)

// checks that the term has type constraint_type
fn check_type(mut self, term_id: TermId, constraint_type: RefinedBase) -> (Self,
↪  Constraint) {
  let term = &self.body.expressions[term_id];
  match term {
    Term::Block {
      statements,
      trailing_expression,
    } => {
      // ... ommited for brevity, but
      // we check that the trailing expression has type constraint_type
      // among other things
    }
    Term::If {
      condition,
      then_branch,
      else_branch,
    } => {
      // ... ommited for brevity, but
      // we check that condition has type bool
      // c1, c2 are the constraints for the then and else branches
      // both must be satisfied
      (fold, Constraint::Conjunction(c1.into(), c2.into()))
    }
    _ => {
      let (fold, t, c) = self.synth_type(term_id);
      // we verify that t <: constraint_type
      let (fold, c2) = fold.subtype(t.as_refined_base(), constraint_type);
      // both c and c2 must be satisfied
      (fold, Constraint::make_conjunction(c, c2))
    }
  }
}
```

While the `synth_type` function determines the type of a term and any associated constraints, `check_type` verifies the type compatibility and ensures adherence to established constraints.

5.3.4 Entailment and Substitution

In the realm of Ekitai's type system, entailment stands as a mechanism to ascertain whether a given constraint can be satisfied within a specific context, from Figure 23.

```
// within a context, checks if a constraint can be satisfied
fn entailment(context: Context, constraint: Constraint) -> bool {
  match context.pop() {
    (_, None) => solve(constraint),
    (tail, Some((path, RefinedType::Fn(_)))) => entailment(tail, constraint),
    (tail, Some((path, RefinedType::Base(RefinedBase { binder, predicate, base }))))
    ↪  => {
      let context_binder = path.as_name();
      let predicate = substitution(binder, context_binder.clone(), predicate);

      entailment(
        tail,
        Constraint::Implication {
          binder: context_binder,
          base,
          antecedent: predicate,
          consequent: constraint.into(),
        },
      )
    }
  }
}

// substitutes all occurrences of old by new in predicate
fn substitution(old: Name, new: Name, predicate: Predicate) -> Predicate
```

The substitution function (subsection 3.1.3) complements the process by replacing instances of one predicate with another, aiding in the seamless integration of types and constraints within varying contexts.

5.3.5 Interfacing with the SMT Solver

To bolster its type-checking prowess, Ekitai integrates with an SMT solver. This solver acts as a decision-making tool, determining the feasibility of constraints.

```
// gets the constraint, flattens it and then pass it to an SMT solver
// to check if the constraint can be satisfied or not
fn solve(constraint: Constraint) -> bool

// flattens a complex constraint into a data structure that the SMT
```

```
// solver can process more easily
fn flatten(constraint: Constraint) -> Vec<FlatImplicationConstraint>

struct FlatImplicationConstraint {
    binders: Vec<(Name, Type)>,
    antecedent: Predicate,
    consequent: Predicate,
}
```

The `solve` function directs the constraint to the SMT solver after a flattening process. This flattened constraint, represented as `FlatImplicationConstraint`, is optimized for the solver, ensuring efficient and accurate type validations.

5.4 THE EKITAI'S LLVM INTERMEDIATE CODE GENERATOR

After we make sure that the program is free from type errors we are able to bring the AST together with all type metadata brought from Ekitai's typechecker we can finally begin the AST's translation to LLVM-IR. The examples in this section where taken from module **`codegen`** found in Appendix A.

The **`fold_binary_expression`** function is a pivotal part of this translation process. This function handles the translation of binary expressions in the Ekitai language into their LLVM equivalents. The function takes in parameters including a potential pointer for indirect values, the operator involved, and the left-hand side (lhs) and right-hand side (rhs) terms. Inside the function, **`fold_expression`** is recursively called on both lhs and rhs to generate their LLVM-IR counterparts. The function then matches the operator to generate the appropriate LLVM instruction, such as **`build_int_add`** for addition operations. If the operation result needs to be stored indirectly, it handles storing this result; otherwise, it returns the computed value directly.

```
fn fold_binary_expression(
  &self,
  indirect_value: Option<PointerValue<'context>>,
  operator: &BinaryOperator,
  lhs: &TermId,
  rhs: &TermId,
) -> Option<Value<'context>> {
  let lhs = self.fold_expression(None, *lhs);
  let rhs = self.fold_expression(None, *rhs);
  let int_value = match operator {
    BinaryOperator::Arithmetic(arithmetic_op) => match arithmetic_op {
      ArithmeticOperator::Add => self.builder.build_int_add(lhs, rhs, ""),
      // ...
    },
```

```
    BinaryOperator::Compare(compare_op) => {
      // ...
    }
  };
  match indirect_value {
    Some(ptr) => {
      self.builder.build_store(ptr, int_value.as_basic_value_enum());
      None
    }
    None => Some(Value::new(ValueKind::Direct, int_value.into())),
  }
}
```

Similarly, the `fold_call_expression` function translates function call expressions. It first retrieves the type of the callee and ensures it is a callable type. The function then allocates space for the return value if it is not directly returned via registers. The actual call instruction is created using `build_call`, and the function handles whether the return value is direct or indirect.

```
fn fold_call_expression(
  &self,
  indirect_value: Option<PointerValue<'context>>,
  callee: &TermId,
  arguments: &[TermId],
) -> Option<Value<'context>> {
  let callee_type = &self.inference.type_of_expression[*callee];
  let callable_definition = match callee_type {
    Type::FunctionDefinition(callable) => callable,
    _ => panic!("call has no callable type."),
  };
  match callable_definition {
    CallableDefinitionId::FunctionDefinition(id) => {
      let function_info = self.function_info_cache.function_info(&id);
      let function_value = self.function_value_cache.llvm_function_value(&id);

      let return_type = function_info.get_return_type(function_value);

      // Allocates the return value if it is not returned directly via registers
      // i.e. returned indirectly through memory
      let indirect_function_return = match function_info.return_kind {
        ValueKind::Direct => None,
        ValueKind::Indirect => match indirect_value {
          Some(ptr) => Some(ptr.as_basic_value_enum()),
          None => Some(
            self.get_alloca_builder()
              .build_alloca(
                BasicType-
                 ↪   Enum::try_from(return_type.into_pointer_type().get_element_type()).unwrap(),
```

```
                "",
              )
              .as_basic_value_enum(),
          ),
        },
      };
      // Creates LLVM values for arguments/parameters whether they are storable
      // in registers or they are stored in memory (and therefore needs alloc/store instructions)
      let arguments = ...

      // Creating LLVM call instruction
      let call_value = self.builder.build_call(
          function_value,
          arguments.map(|x| x.into()).collect::<Vec<_>>().as_slice(),
          "",
      );

      // Returns to the upper level whether the return value is stored in registers
      match indirect_value {
          Some(_) => None,
          None => match function_info.return_kind {
              ValueKind::Indirect => Some(Value::new(
                  ValueKind::Indirect,
                  indirect_function_return.unwrap(),
              )),
              ValueKind::Direct => Some(Value::new(
                  ValueKind::Direct,
                  call_value.try_as_basic_value().unwrap_left(),
              )),
          },
      }
    }
    CallableDefinitionId::ValueConstructor(constructor_id) => {
      // Constructs a ADT value.
    }
  }
}
```

The `fold_if_expression` function handles the translation of conditional expressions. It constructs LLVM basic blocks for the then and else branches as well as the merge block. It creates the conditional and unconditional branch instructions to ensure proper control flow and uses a Phi node to merge the values from both branches if necessary.

```
fn fold_if_expression(
  &self,
```

```
  indirect_value: Option<PointerValue<'context>>,
  condition: &TermId,
  then_branch: &TermId,
  else_branch: &TermId,
) -> Option<Value<'context>> {
  let comparison = self.fold_expression(None, *condition);

  let then_block = self.context.append_basic_block(self.get_owener_function_value(),
  ↪  "then");
  let else_block = self.context.append_basic_block(self.get_owener_function_value(),
  ↪  "else");
  let merge_block = self.context.append_basic_block(self.get_owener_function_value(),
  ↪  "merge");

  // Creates branch instruction to then branch
  self.builder.build_conditional_branch(comparison, then_block, else_block);

  self.builder.position_at_end(then_block);
  let then_value = self.fold_expression(indirect_value, *then_branch);
  let then_block = self.builder.get_insert_block().unwrap();
  // Creates branch instruction to go to the merge block
  self.builder.build_unconditional_branch(merge_block);

  self.builder.position_at_end(else_block);
  let else_value = self.fold_expression(indirect_value, *else_branch);
  let else_block = self.builder.get_insert_block().unwrap();
  // Creates branch instruction to go to the merge block
  self.builder.build_unconditional_branch(merge_block);

  self.builder.position_at_end(merge_block);

  match indirect_value {
    Some(ptr) => None,
    None => {
      let (
        Value { kind: then_kind, value: then_value },
        Value { kind: else_kind, value: else_value },
      ) = (then_value.unwrap(), else_value.unwrap());
      match (then_kind, else_kind) {
        (ValueKind::Direct, ValueKind::Direct) | (ValueKind::Indirect,
        ↪  ValueKind::Indirect) => {
          let phi = self.builder.build_phi(then_value.get_type(), "phi");
          phi.add_incoming(&[(&then_value, then_block), (&else_value, else_block)]);
          Some(Value::new(then_kind, phi.as_basic_value()))
        }
        _ => ...,
      }
    }
```

```
  }
}
```

These examples illustrate the meticulous process of translating high-level Ekitai constructs into LLVM-IR, ensuring that the generated code is both efficient and accurate. The careful handling of direct and indirect values, precise control flow management, and type-aware function calls are fundamental to producing robust and performant intermediate code.

In LLVM-IR, the distinction between direct and indirect values is crucial for efficient memory management and performance optimization. Direct values are those that reside in registers, allowing for fast access and manipulation by the CPU. These are typically used for small, frequently accessed data such as integers or pointers. By storing these values directly in registers, the program can perform computations more quickly, leveraging the high-speed access provided by the CPU's register file.

On the other hand, indirect values are those that reside in memory, such as the stack or heap. These are accessed via pointers, which provide a reference to the actual data location. Indirect values are necessary for larger data structures or when dealing with values that have a longer lifespan than a single function call. For example, local variables of a function are often stored on the stack, allowing them to persist across function calls and be accessed indirectly. This approach is essential for managing larger datasets that cannot fit within the limited register space and for ensuring data integrity across different scopes and function invocations.

The decision to use direct or indirect values impacts the code generation process significantly. Direct values facilitate quick arithmetic and logic operations, while indirect values enable the handling of complex data structures and longer-term data storage. Efficiently managing this distinction allows the compiler to generate optimized code that balances speed and memory usage, ensuring high performance and resource efficiency in the resulting program.

There are two structures responsible to decide if a value will be direct or indirect. They are:

- The `CodeGenTypeCache` structure is responsible for caching LLVM types to ensure efficient reuse throughout the code generation process.

```rust
pub struct CodeGenTypeCache<'db, 'context> {
  db: &'db dyn CodeGenDatabase,
  context: &'context Context,
  target_data: TargetData,
  adt_map: RefCell<HashMap<TypeDefinitionId, (StructType<'context>,
  ↪  TypeInfo<'context>)>>,
  adt_variant_map: RefCell<HashMap<ValueConstructorId, StructType<'context>>>,
}
```

It maintains a mapping between Ekitai types and their corresponding LLVM representations. By caching these types, it avoids the repeated computation of LLVM type representations, which improves the performance and consistency of type handling within the compiler.

- The `CodeGenFunctionInfoCache` structure manages information related to function definitions in the code generation process.

```
struct CodeGenFunctionInfoCache<'db, 'context, 'type_cache> {
  db: &'db dyn CodeGenDatabase,
  context: &'context Context,
  type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
}
```

It caches details about function signatures, return types, and parameter types. This cache allows the code generator to quickly retrieve function-related metadata, ensuring efficient generation of function call instructions and correct handling of function returns.

These structures play a crucial role in optimizing the code generation process by caching frequently used type and function information, thus enhancing the performance and reliability of the Ekitai compiler's backend. They are also the structures responsible to make Ekitai ABI compatible with the C programming language.

5.5 EXAMPLES AND INTERMEDIATE CONSTRUCTS

In this section we will be breathly discussing working examples of the Ekitai compiler.

5.5.1 Identity function with refinements

Take for example the following identity function with refinements written in ekitai:

```
fn id(x: {y: i64 | true}) -> {z: i64 | z == x} {
  x
}
```

Ekitai's type checker will generate the following context to start entailment for the return expression `x`:

```
Context { bindings: [
  ( Path { segments: [Name { id: "id" }] },
    Fn(DependentFunction {
      parameter: (
```

```
        Name { id: "x" },
        RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" }, predicate:
        ↪   Boolean(true) }
      ),
      tail_type: Base(RefinedBase {
        base: Scalar(Integer(I64)), binder: Name { id: "z" },
        predicate: Binary(
          Compare(Equality { negated: false }), Variable(Name { id: "z" }),
          ↪   Variable(Name { id: "x" })
        )
      })
    })
  ),
  ( Path { segments: [Name { id: "x" }] },
    Base(RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" },
    ↪   predicate: Boolean(true) })
  ),
] }
```

The flattened constraint generate is:

```
Context { bindings: [
  ( Path { segments: [Name { id: "id" }] },
    Fn(DependentFunction {
      parameter: (
        Name { id: "x" },
        RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" }, predicate:
        ↪   Boolean(true) }
      ),
      tail_type: Base(RefinedBase {
        base: Scalar(Integer(I64)), binder: Name { id: "z" },
        predicate: Binary(
          Compare(Equality { negated: false }), Variable(Name { id: "z" }),
          ↪   Variable(Name { id: "x" })
        )
      })
    })
  ),
  ( Path { segments: [Name { id: "x" }] },
    Base(RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" },
    ↪   predicate: Boolean(true) })
  ),
] }
```

After translation to Z3:

```
(declare-fun x () Int)
(declare-fun y () Int)
```

```
(assert (and true true (= y x) true true (not (= y x))))

Result: Unsat
```

The result Unsat means that there is no possible values for the types in the program that would fail the constraint.

After proof of the absence of type errors, Ekitai's compiler generates the following LLVM-IR:

```
; ModuleID = 'ekitai_module'
source_filename = "ekitai_module"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

define i64 @id(i64 %x) {
  ret i64 %x
}
```

5.5.2 Absolute value function with refinements

Take for exemaple this function for absolute integer values with refinements:

```
fn abs_liquid(x: {y: i64 | true}) -> {z: i64 | z >= 0} {
  if x > 0 {
    x
  } else {
    -x
  }
}
```

Ekitai's type checker will check for entailment in multiple points durring type-checking. In the following code, we can see the starting context for the body of the `abs_liquid` function:

```
  Context: Context {
    bindings: [
        (
            Path {
                segments: [
                    Name {
                        id: "abs_liquid",
                    },
                ],
            },
            Fn(
                DependentFunction {
```

```
                    parameter: (
                        Name {
                            id: "x",
                        },
                        RefinedBase {
                            base: Scalar(
                                Integer(
                                    I64,
                                ),
                            ),
                            binder: Name {
                                id: "y",
                            },
                            predicate: Boolean(
                                true,
                            ),
                        },
                    ),
                    tail_type: Base(
                        RefinedBase {
                            base: Scalar(
                                Integer(
                                    I64,
                                ),
                            ),
                            binder: Name {
                                id: "z",
                            },
                            predicate: Binary(
                                Compare(
                                    Order {
                                        ordering: Greater,
                                        strict: false,
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "z",
                                    },
                                ),
                                Integer(
                                    0,
                                ),
                            ),
                        },
                    ),
                },
            ),
```

```
        ),
        (
            Path {
                segments: [
                    Name {
                        id: "x",
                    },
                ],
            },
            Base(
                RefinedBase {
                    base: Scalar(
                        Integer(
                            I64,
                        ),
                    ),
                    binder: Name {
                        id: "y",
                    },
                    predicate: Boolean(
                        true,
                    ),
                },
            ),
        ),
        (
            Path {
                segments: [
                    Name {
                        id: "__arg0",
                    },
                ],
            },
            Base(
                RefinedBase {
                    base: Scalar(
                        Integer(
                            I64,
                        ),
                    ),
                    binder: Name {
                        id: "y",
                    },
                    predicate: Binary(
                        Logic(
                            And,
                        ),
                        Boolean(
```

```
                            true,
                        ),
                        Binary(
                            Compare(
                                Equality {
                                    negated: false,
                                },
                            ),
                            Variable(
                                Name {
                                    id: "y",
                                },
                            ),
                            Variable(
                                Name {
                                    id: "x",
                                },
                            ),
                        ),
                    ),
                },
            ),
        ),
        (
            Path {
                segments: [
                    Name {
                        id: "__arg1",
                    },
                ],
            },
            Base(
                RefinedBase {
                    base: Scalar(
                        Integer(
                            I64,
                        ),
                    ),
                    binder: Name {
                        id: "lit",
                    },
                    predicate: Binary(
                        Compare(
                            Equality {
                                negated: false,
                            },
                        ),
                        Variable(
```

```
                            Name {
                                id: "lit",
                            },
                        ),
                        Integer(
                            0,
                        ),
                    ),
                },
            ),
        ),
        (
            Path {
                segments: [
                    Name {
                        id: "__arg2",
                    },
                ],
            },
            Base(
                RefinedBase {
                    base: Scalar(
                        Boolean,
                    ),
                    binder: Name {
                        id: "ret",
                    },
                    predicate: Binary(
                        Compare(
                            Equality {
                                negated: false,
                            },
                        ),
                        Variable(
                            Name {
                                id: "ret",
                            },
                        ),
                        Binary(
                            Compare(
                                Order {
                                    ordering: Greater,
                                    strict: true,
                                },
                            ),
                            Variable(
                                Name {
                                    id: "__arg0",
```

```
                                },
                            ),
                            Variable(
                                Name {
                                    id: "__arg1",
                                },
                            ),
                        ),
                    ),
                },
            ),
        ),
        (
            Path {
                segments: [
                    Name {
                        id: "__arg3",
                    },
                ],
            },
            Base(
                RefinedBase {
                    base: Scalar(
                        Integer(
                            I64,
                        ),
                    ),
                    binder: Name {
                        id: "y",
                    },
                    predicate: Binary(
                        Logic(
                            And,
                        ),
                        Boolean(
                            true,
                        ),
                        Binary(
                            Compare(
                                Equality {
                                    negated: false,
                                },
                            ),
                            Variable(
                                Name {
                                    id: "y",
                                },
                            ),
```

```
                        Variable(
                            Name {
                                id: "x",
                            },
                        ),
                    ),
                ),
            },
        ),
    ),
],
}
```

Followed by the flattened constraint generated from the context:

```
Constraint: Implication {
    binder: Name {
        id: "x",
    },
    base: Scalar(
        Integer(
            I64,
        ),
    ),
    antecedent: Boolean(
        true,
    ),
    consequent: Conjunction(
        Implication {
            binder: Name {
                id: "__arg0",
            },
            base: Scalar(
                Integer(
                    I64,
                ),
            ),
            antecedent: Binary(
                Logic(
                    And,
                ),
                Boolean(
                    true,
                ),
                Binary(
                    Compare(
                        Equality {
                            negated: false,
```

```
                        },
                    ),
                    Variable(
                        Name {
                            id: "__arg0",
                        },
                    ),
                    Variable(
                        Name {
                            id: "x",
                        },
                    ),
                ),
            ),
            consequent: Implication {
                binder: Name {
                    id: "__arg1",
                },
                base: Scalar(
                    Integer(
                        I64,
                    ),
                ),
                antecedent: Binary(
                    Compare(
                        Equality {
                            negated: false,
                        },
                    ),
                    Variable(
                        Name {
                            id: "__arg1",
                        },
                    ),
                    Integer(
                        0,
                    ),
                ),
                consequent: Conjunction(
                    Implication {
                        binder: Name {
                            id: "y",
                        },
                        base: Scalar(
                            Integer(
                                I64,
                            ),
                        ),
```

```
                        antecedent: Binary(
                            Logic(
                                And,
                            ),
                            Binary(
                                Logic(
                                    And,
                                ),
                                Boolean(
                                    true,
                                ),
                                Binary(
                                    Compare(
                                        Equality {
                                            negated: false,
                                        },
                                    ),
                                    Variable(
                                        Name {
                                            id: "y",
                                        },
                                    ),
                                    Variable(
                                        Name {
                                            id: "x",
                                        },
                                    ),
                                ),
                            ),
                            Binary(
                                Compare(
                                    Equality {
                                        negated: false,
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "y",
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "__arg0",
                                    },
                                ),
                            ),
                        ),
                        consequent: Predicate(
```

```
                            Boolean(
                                true,
                            ),
                        ),
                    },
                    Implication {
                        binder: Name {
                            id: "lit",
                        },
                        base: Scalar(
                            Integer(
                                I64,
                            ),
                        ),
                        antecedent: Binary(
                            Logic(
                                And,
                            ),
                            Binary(
                                Compare(
                                    Equality {
                                        negated: false,
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "lit",
                                    },
                                ),
                                Integer(
                                    0,
                                ),
                            ),
                            Binary(
                                Compare(
                                    Equality {
                                        negated: false,
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "lit",
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "__arg1",
                                    },
```

```
                                ),
                            ),
                        ),
                        consequent: Predicate(
                            Boolean(
                                true,
                            ),
                        ),
                    },
                ),
            },
        },
        Implication {
            binder: Name {
                id: "__arg2",
            },
            base: Scalar(
                Boolean,
            ),
            antecedent: Binary(
                Compare(
                    Equality {
                        negated: false,
                    },
                ),
                Variable(
                    Name {
                        id: "__arg2",
                    },
                ),
                Binary(
                    Compare(
                        Order {
                            ordering: Greater,
                            strict: true,
                        },
                    ),
                    Variable(
                        Name {
                            id: "__arg0",
                        },
                    ),
                    Variable(
                        Name {
                            id: "__arg1",
                        },
                    ),
                ),
```

```
            ),
            consequent: Conjunction(
                Implication {
                    binder: Name {
                        id: "__fresh0",
                    },
                    base: Scalar(
                        Boolean,
                    ),
                    antecedent: Variable(
                        Name {
                            id: "__arg2",
                        },
                    ),
                    consequent: Implication {
                        binder: Name {
                            id: "y",
                        },
                        base: Scalar(
                            Integer(
                                I64,
                            ),
                        ),
                        antecedent: Binary(
                            Logic(
                                And,
                            ),
                            Boolean(
                                true,
                            ),
                            Binary(
                                Compare(
                                    Equality {
                                        negated: false,
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "y",
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "x",
                                    },
                                ),
                            ),
                        ),
```

```
                        consequent: Predicate(
                            Binary(
                                Compare(
                                    Order {
                                        ordering: Greater,
                                        strict: false,
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "y",
                                    },
                                ),
                                Integer(
                                    0,
                                ),
                            ),
                        ),
                    },
                },
                Implication {
                    binder: Name {
                        id: "__fresh0",
                    },
                    base: Scalar(
                        Boolean,
                    ),
                    antecedent: Unary(
                        Negation,
                        Variable(
                            Name {
                                id: "__arg2",
                            },
                        ),
                    ),
                    consequent: Implication {
                        binder: Name {
                            id: "__arg3",
                        },
                        base: Scalar(
                            Integer(
                                I64,
                            ),
                        ),
                        antecedent: Binary(
                            Logic(
                                And,
                            ),
```

```
                            Boolean(
                                true,
                            ),
                            Binary(
                                Compare(
                                    Equality {
                                        negated: false,
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "__arg3",
                                    },
                                ),
                                Variable(
                                    Name {
                                        id: "x",
                                    },
                                ),
                            ),
                        ),
                        consequent: Conjunction(
                            Implication {
                                binder: Name {
                                    id: "y",
                                },
                                base: Scalar(
                                    Integer(
                                        I64,
                                    ),
                                ),
                                antecedent: Binary(
                                    Logic(
                                        And,
                                    ),
                                    Binary(
                                        Logic(
                                            And,
                                        ),
                                        Boolean(
                                            true,
                                        ),
                                        Binary(
                                            Compare(
                                                Equality {
                                                    negated: false,
                                                },
                                            ),
```

```
                                            Variable(
                                                Name {
                                                    id: "y",
                                                },
                                            ),
                                            Variable(
                                                Name {
                                                    id: "x",
                                                },
                                            ),
                                        ),
                                    ),
                                    Binary(
                                        Compare(
                                            Equality {
                                                negated: false,
                                            },
                                        ),
                                        Variable(
                                            Name {
                                                id: "y",
                                            },
                                        ),
                                        Variable(
                                            Name {
                                                id: "__arg3",
                                            },
                                        ),
                                    ),
                                ),
                                consequent: Predicate(
                                    Boolean(
                                        true,
                                    ),
                                ),
                            },
                            Implication {
                                binder: Name {
                                    id: "ret",
                                },
                                base: Scalar(
                                    Integer(
                                        I64,
                                    ),
                                ),
                                antecedent: Binary(
                                    Compare(
                                        Equality {
```

```
                                            negated: false,
                                        },
                                    ),
                                    Variable(
                                        Name {
                                            id: "ret",
                                        },
                                    ),
                                    Unary(
                                        Minus,
                                        Variable(
                                            Name {
                                                id: "__arg3",
                                            },
                                        ),
                                    ),
                                ),
                                consequent: Predicate(
                                    Binary(
                                        Compare(
                                            Order {
                                                ordering: Greater,
                                                strict: false,
                                            },
                                        ),
                                        Variable(
                                            Name {
                                                id: "ret",
                                            },
                                        ),
                                        Integer(
                                            0,
                                        ),
                                    ),
                                ),
                            },
                        ),
                    },
                },
            ),
        },
    ),
}
```

During typecheking of the `abs_liquid` function’s body, there will be multiple calls to entailment. We can see the calls made in the following code:

```
Solver:
```

```
(declare-fun x () Int)
(declare-fun __arg0 () Int)
(declare-fun __arg1 () Int)
(declare-fun __arg2 () Bool)
(declare-fun ret () Bool)
(assert (and true
     (= ret (> __arg0 __arg1))
     (= ret __arg2)
     (= __arg2 (> __arg0 __arg1))
     (= __arg1 0)
     true
     (= __arg0 x)
     true
     (not true)))

Result: Unsat
Solver:
(declare-fun x () Int)
(declare-fun __arg0 () Int)
(declare-fun __arg1 () Int)
(declare-fun __arg2 () Bool)
(declare-fun __arg3 () Int)
(declare-fun y () Int)
(assert (and true
      true
      (= y x)
      (= y __arg0)
      (= __arg1 0)
      (and true (= __arg0 x))
      true
      true
      (= __arg3 x)
      (= __arg2 (> __arg0 __arg1))
      (= __arg1 0)
      (and true (= __arg0 x))
      true
      (not true)))

Result: Unsat
Solver:
(declare-fun x () Int)
(declare-fun __arg0 () Int)
(declare-fun __arg1 () Int)
(declare-fun __arg2 () Bool)
(declare-fun __arg3 () Int)
(declare-fun lit () Int)
(assert (and true
      (= lit 0)
```

```
      (= lit __arg1)
      (= __arg1 0)
      (and true (= __arg0 x))
      true
      true
      (= __arg3 x)
      (= __arg2 (> __arg0 __arg1))
      (= __arg1 0)
      (and true (= __arg0 x))
      true
      (not true)))

Result: Unsat
Solver:
(declare-fun y () Int)
(declare-fun x () Int)
(declare-fun __arg0 () Int)
(declare-fun __arg1 () Int)
(declare-fun __arg2 () Bool)
(declare-fun __arg3 () Int)
(assert (and true
      true
      (= y x)
      __arg2
      (= __arg2 (> __arg0 __arg1))
      true
      true
      (= __arg3 x)
      (= __arg2 (> __arg0 __arg1))
      (= __arg1 0)
      true
      (= __arg0 x)
      true
      (not (>= y 0))))

Result: Unsat
Solver:
(declare-fun x () Int)
(declare-fun __arg0 () Int)
(declare-fun __arg1 () Int)
(declare-fun __arg2 () Bool)
(declare-fun __arg3 () Int)
(declare-fun y () Int)
(assert (and true
      true
      (= y x)
      (= y __arg3)
      (and true (= __arg3 x))
```

```
      (not __arg2)
      (= __arg2 (> __arg0 __arg1))
      true
      (and true (= __arg3 x))
      (= __arg2 (> __arg0 __arg1))
      (= __arg1 0)
      true
      (= __arg0 x)
      true
      (not true)))

Result: Unsat
Solver:
(declare-fun ret () Int)
(declare-fun x () Int)
(declare-fun __arg0 () Int)
(declare-fun __arg1 () Int)
(declare-fun __arg2 () Bool)
(declare-fun __arg3 () Int)
(assert (and true
      (= ret (- __arg3))
      (and true (= __arg3 x))
      (not __arg2)
      (= __arg2 (> __arg0 __arg1))
      true
      (and true (= __arg3 x))
      (= __arg2 (> __arg0 __arg1))
      (= __arg1 0)
      true
      (= __arg0 x)
      true
      (not (>= ret 0))))

Result: Unsat
```

After Ekitai's compiler ensures the absence of type errors, the following LLVM-IR code is generated:

```
; ModuleID = 'ekitai_module'
source_filename = "ekitai_module"
target datalayout =
↪   "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

define i64 @abs_liquid(i64 %x) {
  %1 = icmp sgt i64 %x, 0
  br i1 %1, label %then, label %else

```

```
then:                                             ; preds = %0
  br label %merge

else:                                             ; preds = %0
  %2 = sub i64 0, %x
  br label %merge

merge:                                            ; preds = %else, %then
  %phi = phi i64 [ %x, %then ], [ %2, %else ]
  ret i64 %phi
}
```

6 CONCLUSION

The Ekitai language project demonstrates the successful integration of refinement types with an LLVM-IR front-end implementation. This integration allows for more robust type checking and verification, providing both safety and performance optimizations in the generated code. By leveraging refinement types, Ekitai ensures that programs adhere to stricter constraints, thus reducing runtime errors and improving reliability.

The detailed exploration of Ekitai's lexer, parser, type system, and LLVM intermediate code generator highlights the complex interplay between various components in a modern compiler. The design and implementation choices, such as the use of SMT solvers for type verification and the handling of direct and indirect values in code generation, underscore the importance of efficient and accurate translation from high-level language constructs to low-level machine instructions.

6.1 FINAL CONSIDERATIONS

The development of Ekitai marks a significant step towards safer and more efficient programming languages. By integrating refinement types, Ekitai not only enhances type safety but also opens up new avenues for compiler optimizations. The meticulous design of its components, from lexical analysis to code generation, ensures that the language can be extended and adapted to meet future needs.

The integration with LLVM provides a solid foundation for generating highly optimized machine code, benefiting from LLVM's extensive suite of optimizations. Additionally, the choice to implement Ekitai in Rust adds an extra layer of safety and performance, leveraging Rust's strong type system and ownership model to prevent common programming errors.

6.2 FUTURE WORK

While Ekitai represents a robust foundation, there are several areas for future work and improvement:

- **Predicate-Based Optimizations**: Leveraging the predicates defined in refinement types for advanced optimizations in the Abstract Syntax Tree (AST) and LLVM code generation phase. This could involve:
    - **Bounds Checking Elimination**: Using predicates to prove at compile time that certain array accesses are always within bounds, thus eliminating the need for runtime checks.

  - **Nullability Checks Removal**: Ensuring through predicates that certain variables can never be null, allowing the removal of redundant null checks.
  - **Specialized Code Paths**: Generating specialized code paths for different refined types, optimizing the code for specific cases known at compile time.

- **Extended Refinement Type System**: Enhancing the refinement type system to support more complex constraints and type relationships could provide additional safety guarantees. This could include support for dependent types or more sophisticated constraint solvers.

- **Improved Error Reporting**: Enhancing the compiler's error reporting mechanisms to provide more informative and user-friendly messages can significantly improve the developer experience. This includes better integration of refinement type errors and suggestions for fixing them.

- **Benchmarking and Real-World Testing**: Conducting extensive benchmarking and real-world testing with a variety of applications will help identify performance bottlenecks and areas for further optimization. This empirical data can guide future enhancements to both the compiler and the language itself.

- **User and Community Feedback**: Actively seeking feedback from users and the developer community can provide valuable insights into the practical usability of Ekitai. This feedback can drive the prioritization of new features and improvements based on actual user needs.

By addressing these areas, Ekitai can continue to evolve and provide a powerful, safe, and efficient programming language for a wide range of applications.

A EKITAI'S IMPLEMENTATION

The Ekitai's implementation in Rust spans more than 12000 lines of code across 121 files and hence will not be fully anexed to this document. Please refere to the code's repository hosted at `https://github.com/tarberd/ekitai/`.

A.1 SUBMODULE HIR::LIQUID

```rust
use core::panic;
use std::vec;

use la_arena::Idx;
use z3::ast::{Ast, Bool, Int};

use crate::{
    check::{
        type_inference::{InferenceResult, TypeReferenceResolver},
        IntegerKind, ScalarType, Type,
    },
    semantic_ir::{
        definition_map::{FunctionDefinitionData, FunctionDefinitionId},
        intrinsic::BuiltinInteger,
        name::Name,
        path::Path,
        path_resolver::Resolver,
        refinement::{Predicate, UnaryOperator},
        term::{
            ArithmeticOperator, BinaryOperator, Body, CompareOperator, Literal,
            ↪ LogicOperator,
            Pattern, PatternId, Statement, Term, TermId, UnaryOperator as TermUnaryOp,
        },
        type_reference::TypeReference,
    },
    HirDatabase,
};

#[derive(Debug, Clone)]
struct RefinedBase {
    pub base: Type,
    pub binder: Name,
    pub predicate: Predicate,
}

#[derive(Debug, Clone)]
struct DependentFunction {
    pub parameter: (Name, RefinedBase),
```

```rust
    pub tail_type: Box<RefinedType>,
}

#[derive(Debug, Clone)]
enum RefinedType {
    Base(RefinedBase),
    Fn(DependentFunction),
}

impl RefinedType {
    fn as_refined_base(self) -> RefinedBase {
        match self {
            RefinedType::Base(base) => base,
            _ => panic!("Not a RefinedBase."),
        }
    }

    fn as_dependent_function(self) -> DependentFunction {
        match self {
            RefinedType::Fn(func) => func,
            _ => panic!("Not a DependentFunction."),
        }
    }
}

impl From<RefinedBase> for RefinedType {
    fn from(base: RefinedBase) -> Self {
        Self::Base(base)
    }
}

impl From<DependentFunction> for RefinedType {
    fn from(fun: DependentFunction) -> Self {
        Self::Fn(fun)
    }
}

#[derive(Debug, Clone)]
struct Context {
    bindings: Vec<(Path, RefinedType)>,
}

impl Context {
    fn get(&self, path: &Path) -> Option<RefinedType> {
        self.bindings
            .iter()
            .find(|(to_find, _)| path == to_find)
            .map(|(_, ty)| ty.clone())
```

```
    }

    fn pop(mut self) -> (Self, Option<(Path, RefinedType)>) {
        let opt = self.bindings.pop();
        (self, opt)
    }
}

impl FromIterator<(Path, RefinedType)> for Context {
    fn from_iter<T: IntoIterator<Item = (Path, RefinedType)>>(iter: T) -> Self {
        let bindings = iter.into_iter().collect();
        Self { bindings }
    }
}

#[derive(Debug)]
enum Constraint {
    Predicate(Predicate),
    Implication {
        binder: Name,
        base: Type,
        antecedent: Predicate,
        consequent: Box<Self>,
    },
    Conjunction(Box<Self>, Box<Self>),
}

impl Constraint {
    fn make_conjunction(left: Option<Self>, right: Self) -> Self {
        match left {
            Some(left) => Self::Conjunction(left.into(), right.into()),
            None => right,
        }
    }
}

fn something(
    db: &dyn HirDatabase,
    resolver: &Resolver,
    type_reference: &TypeReference,
) -> RefinedBase {
    match type_reference {
        TypeReference::Path(_) => todo!(),
        TypeReference::Refinement(inner, binder, predicate) => {
            let type_ref_resolver = TypeReferenceResolver::new(db, resolver);
           let inner_type = type_ref_resolver.resolve_type_reference(&inner).unwrap();
            RefinedBase {
                base: inner_type,
```

```
                binder: binder.clone(),
                predicate: predicate.clone(),
            }
        }
        TypeReference::Pointer(_) => todo!(),
    }
}

pub fn check_abstraction(db: &dyn HirDatabase, function_id: FunctionDefinitionId) -> bool {
    let context = {
        db.source_file_definitions_map()
            .root_module_item_scope()
            .iter_function_locations()
            .map(|fid| {
                let FunctionDefinitionData { name, .. } = db.function_definition_data(*fid);
                (
                    Path {
                        segments: vec![name],
                    },
                    make_function_type(db, *fid),
                )
            })
            .collect()
    };

    let function_type = make_function_type(db, function_id);
    let body = db.body_of_definition(function_id);

    let (context, constraint) = match function_type {
        RefinedType::Base(base) => {
            let (Fold { context, .. }, constraint) = Fold {
                body: &body,
                context,
                inference: db.infer_body_expression_types(function_id),
                fresh_var_counter: 0,
            }
            .check_type(body.root_expression, base);
            (context, constraint)
        }
        RefinedType::Fn(depfn) => {
            let (Fold { context, .. }, constraint) = Fold {
                body: &body,
                context,
                inference: db.infer_body_expression_types(function_id),
                fresh_var_counter: 0,
            }
```

```
            .check_abstraction_type(depfn, body.root_expression);
            (context, constraint)
        }
    };

    println!("Context: {:?}", context);
    println!("Constraint: {:?}", constraint);
    entailment(context, constraint)
}

fn make_function_type(db: &dyn HirDatabase, function_id: FunctionDefinitionId) -> RefinedType {
    let resolver = Resolver::new_for_function(db.upcast(), function_id);
    let function = db.function_definition_data(function_id);
    let output_type = something(db, &resolver, &function.return_type);

    let body = db.body_of_definition(function_id.into());
    let param_types = function
        .parameter_types
        .iter()
        .map(|reference| something(db, &resolver, reference));

    let function_type = body
        .parameters
        .iter()
        .map(|id| {
            let pat = &body.patterns[*id];
            match pat {
               Pattern::Deconstructor(_, _) => panic!("no deconstructor on parameter"),
                Pattern::Bind(name) => name,
            }
        })
        .cloned()
        .zip(param_types.clone())
        .rfold(
            RefinedType::Base(output_type),
            |tail_type, (param_name, param_type)| {
                DependentFunction {
                    parameter: (param_name, param_type),
                    tail_type: tail_type.into(),
                }
                .into()
            },
        );

    function_type
}

```

```
fn entailment(context: Context, constraint: Constraint) -> bool {
    match context.pop() {
        (_, None) => solve(constraint),
        (tail, Some((_path, RefinedType::Fn(_)))) => entailment(tail, constraint),
        (tail, Some((path, RefinedType::Base(refinement)))) => {
            let RefinedBase {
                binder,
                predicate,
                base,
            } = refinement;

            let context_binder = path.as_name();
            let predicate = substitution(binder, context_binder.clone(), predicate);

            entailment(
                tail,
                Constraint::Implication {
                    binder: context_binder,
                    base,
                    antecedent: predicate,
                    consequent: constraint.into(),
                },
            )
        }
    }
}

#[derive(Debug, Clone)]
enum Z3Predicate<'ctx> {
    Bool(Bool<'ctx>),
    Int(Int<'ctx>),
}

impl<'ctx> From<Bool<'ctx>> for Z3Predicate<'ctx> {
    fn from(from: Bool<'ctx>) -> Self {
        Z3Predicate::Bool(from)
    }
}

impl<'ctx> From<Int<'ctx>> for Z3Predicate<'ctx> {
    fn from(from: Int<'ctx>) -> Self {
        Z3Predicate::Int(from)
    }
}

fn solve(constraint: Constraint) -> bool {
    let flattened_constraint = flatten(constraint);

```

```
    let solver_config = z3::Config::new();
    let solver_context = z3::Context::new(&solver_config);
    let solver = z3::Solver::new(&solver_context);

    for constraint in flattened_constraint {
        let constraint = lower(&solver_context, constraint);

        solver.push();
        solver.assert(&constraint);
        println!("Solver:\n{solver}");
        let result = solver.check();
        println!("Result: {result:?}");
        solver.pop(1);

        match result {
            z3::SatResult::Unsat => continue,
            z3::SatResult::Unknown => return false,
            z3::SatResult::Sat => return false,
        };
    }

    true
}

fn lower(context: &z3::Context, constraint: FlatImplicationConstraint) -> Bool {
    let FlatImplicationConstraint {
        binders,
        antecedent,
        consequent,
    } = constraint;

    let variables: Vec<(Name, Z3Predicate)> = binders
        .into_iter()
        .map(|(name, base)| match base {
            Type::AbstractDataType(_) => todo!("no abstract data types supported in
            ↪   liquid terms"),
            Type::FunctionDefinition(_) => todo!("no function data types in liquid
            ↪   terms"),
            Type::Pointer(_) => todo!("no pointer type in liquid terms"),
            Type::Scalar(scalar) => match scalar {
                crate::check::ScalarType::Integer(_) => (
                    name.clone(),
                    Int::new_const(context, name.id.as_str()).into(),
                ),
                crate::check::ScalarType::Boolean => (
                    name.clone(),
                    Bool::new_const(context, name.id.as_str()).into(),
                ),
```

```rust
            },
        })
        .collect();

    let antecedent = lower_predicate(context, &variables, antecedent);
    let consequent = lower_predicate(context, &variables, consequent);

    match (antecedent, consequent) {
        (Z3Predicate::Bool(prepo), Z3Predicate::Bool(conse)) => {
            Bool::and(context, &[&prepo, &conse.not()])
        }
        _ => todo!(),
    }
}

fn lower_predicate<'ctx>(
    context: &'ctx z3::Context,
    variables: &Vec<(Name, Z3Predicate<'ctx>)>,
    predicate: Predicate,
) -> Z3Predicate<'ctx> {
    match predicate {
        Predicate::Variable(name) => {
            let x = variables
                .iter()
                .find(|(to_find, _)| name == *to_find)
                .map(|(_, ty)| ty)
                .expect("liquid variable not in context");
            x.clone()
        }
        Predicate::Boolean(value) => Bool::from_bool(context, value).into(),
        Predicate::Integer(value) => Int::from_u64(context, value as u64).into(),
        Predicate::Binary(op, lhs, rhs) => {
            let lhs = lower_predicate(context, variables, *lhs);
            let rhs = lower_predicate(context, variables, *rhs);
            match op {
                BinaryOperator::Arithmetic(arith_op) => {
                    let (lhs, rhs) = match (lhs, rhs) {
                        (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => (lhs, rhs),
                        _ => panic!(),
                    };
                    match arith_op {
                        ArithmeticOperator::Add => Int::add(context, &[&lhs, &rhs]),
                        ArithmeticOperator::Sub => Int::sub(context, &[&lhs, &rhs]),
                        ArithmeticOperator::Div => lhs.div(&rhs),
                        ArithmeticOperator::Mul => Int::mul(context, &[&lhs, &rhs]),
                        ArithmeticOperator::Rem => lhs.rem(&rhs),
                    }
                    .into()
```

```
                }
                BinaryOperator::Logic(op) => {
                    let (lhs, rhs) = match (lhs, rhs) {
                       (Z3Predicate::Bool(lhs), Z3Predicate::Bool(rhs)) => (lhs, rhs),
                        _ => todo!(),
                    };
                    match op {
                       LogicOperator::And => Bool::and(context, &[&lhs, &rhs]).into(),
                        LogicOperator::Or => Bool::or(context, &[&lhs, &rhs]).into(),
                    }
                }
                BinaryOperator::Compare(compare) => match compare {
                    CompareOperator::Equality { negated } => match (lhs, rhs) {
                        (Z3Predicate::Bool(lhs), Z3Predicate::Bool(rhs)) => match
                        ↪   negated {
                            false => lhs._eq(&rhs).into(),
                            true => Bool::distinct(context, &[&lhs, &rhs]).into(),
                        },
                      (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => match negated {
                            false => lhs._eq(&rhs).into(),
                            true => Int::distinct(context, &[&lhs, &rhs]).into(),
                        },
                        _ => panic!("mismatch types in z3"),
                    },
                    CompareOperator::Order { ordering, strict } => match ordering {
                        crate::semantic_ir::term::Ordering::Less => match (lhs, rhs) {
                            (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => match
                            ↪   strict {
                                true => lhs.lt(&rhs).into(),
                                false => lhs.le(&rhs).into(),
                            },
                            _ => todo!(),
                        },
                      crate::semantic_ir::term::Ordering::Greater => match (lhs, rhs) {
                            (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => match
                            ↪   strict {
                                true => lhs.gt(&rhs).into(),
                                false => lhs.ge(&rhs).into(),
                            },
                            _ => todo!(),
                        },
                    },
                },
            }
        }
        Predicate::Unary(op, predicate) => {
            let predicate = lower_predicate(context, variables, *predicate);
            match (op, predicate) {
```

```
                (UnaryOperator::Minus, Z3Predicate::Int(int)) =>
                ↪  int.unary_minus().into(),
                (UnaryOperator::Negation, Z3Predicate::Bool(boolean)) =>
                ↪  boolean.not().into(),
                (UnaryOperator::Minus, Z3Predicate::Bool(_)) => todo!(),
                (UnaryOperator::Negation, Z3Predicate::Int(_)) => todo!(),
            }
        }
    }
}

struct FlatImplicationConstraint {
    binders: Vec<(Name, Type)>,
    antecedent: Predicate,
    consequent: Predicate,
}

fn flatten(constraint: Constraint) -> Vec<FlatImplicationConstraint> {
    flatten_fold(Vec::new(), constraint)
}

fn flatten_fold(
    mut flattened: Vec<FlatImplicationConstraint>,
    constraint: Constraint,
) -> Vec<FlatImplicationConstraint> {
    match constraint {
        Constraint::Predicate(predicate) => {
            flattened.push(FlatImplicationConstraint {
                binders: vec![],
                antecedent: Predicate::Boolean(true),
                consequent: predicate,
            });
            flattened
        }
        Constraint::Implication {
            binder,
            base,
            antecedent,
            consequent,
        } => {
            flattened.extend(flatten(*consequent).into_iter().map(
                |FlatImplicationConstraint {
                      mut binders,
                      antecedent: sub_antecedent,
                      consequent,
                  }| {
                    let binders = {
                        binders.push((binder.clone(), base.clone()));
```

```
                        binders
                    };
                    let antecedent = Predicate::Binary(
                        BinaryOperator::Logic(LogicOperator::And),
                        sub_antecedent.into(),
                        antecedent.clone().into(),
                    );
                    FlatImplicationConstraint {
                        binders,
                        antecedent,
                        consequent,
                    }
                },
            ));
            flattened
        }
        Constraint::Conjunction(first, second) => {
            let flattened = flatten_fold(flattened, *first);
            flatten_fold(flattened, *second)
        }
    }
}

struct Fold<'a> {
    body: &'a Body,
    context: Context,
    inference: InferenceResult,
    fresh_var_counter: usize,
}

impl<'a> Fold<'a> {
    fn subtype(self, lesser: RefinedBase, greater: RefinedBase) -> (Self, Constraint) {
        let RefinedBase {
            base: lesser_base,
            binder: lesser_binder,
            predicate: lesser_predicate,
        } = lesser;
        let RefinedBase {
            base: greater_base,
            binder: greater_binder,
            predicate: greater_predicate,
        } = greater;

        if lesser_base != greater_base {
            panic!("missmatch base types: {lesser_base:?} <: {greater_base:?}")
        }

        let constraint = Constraint::Implication {
```

```
                binder: lesser_binder.clone(),
                base: lesser_base,
                antecedent: lesser_predicate,
                consequent: Constraint::Predicate(substitution(
                    greater_binder,
                    lesser_binder,
                    greater_predicate,
                ))
                .into(),
            };

            (self, constraint)
        }

        fn check_abstraction_type(
            mut self,
            constraint_type: DependentFunction,
            body_term: TermId,
        ) -> (Self, Constraint) {
            let DependentFunction {
                parameter: (arg_name, arg_ty),
                tail_type,
            } = constraint_type;
            //add to context
            let arg_path = Path {
                segments: vec![arg_name.clone()],
            };
            self.context
                .bindings
                .push((arg_path, arg_ty.clone().into()));
            //check
            let (fold, constraint) = match *tail_type {
                RefinedType::Base(base) => self.check_type(body_term, base),
                RefinedType::Fn(depfn) => self.check_abstraction_type(depfn, body_term),
            };
            //implication constraint
            let constraint = implication_constraint(arg_name, arg_ty, Some(constraint));
            (fold, constraint)
        }

        fn check_type(mut self, term_id: TermId, constraint_type: RefinedBase) -> (Self,
        ↪  Constraint) {
            let term = &self.body.expressions[term_id];
            match term {
                Term::Block {
                    statements,
                    trailing_expression,
                } => {
```

```rust
    let (fold, constraints) =
        statements
            .iter()
            .fold(
                (self, vec![]),
                |(fold, mut constraints), statement| match statement {
                    Statement::Let(pattern, init_term) => {
                        let (mut fold, ty, c) = fold.synth_type(*init_term);
                        let pattern = &fold.body.patterns[*pattern];
                        let (path, name) = match pattern {
                            Pattern::Bind(name) => (
                                Path {
                                    segments: vec![name.clone()],
                                },
                                name.clone(),
                            ),
                            _ => todo!(),
                        };
                        fold.context.bindings.push((path, ty.clone()));
                        constraints.push((name, ty, c));
                        (fold, constraints)
                    }
                    Statement::Expression(_) => todo!(),
                },
            );
    let (fold, c) = fold.check_type(*trailing_expression, constraint_type);
    let constraint = constraints.into_iter().rfold(
        Some(c),
        |inner_constraint, (name, ty, outer_constraint)| {
            let implication_constraint =
                implication_constraint(name, ty.as_refined_base(),
                ↪   inner_constraint);
            Some(Constraint::make_conjunction(
                outer_constraint,
                implication_constraint,
            ))
        },
    );
    (fold, constraint.unwrap())
}
Term::If {
    condition,
    then_branch,
    else_branch,
} => {
    let fresh_var = self.make_fresh_var();
    let (fold, constraint) = self.check_type(
        *condition,
```

```
                    RefinedBase {
                        base: Type::Scalar(ScalarType::Boolean),
                        binder: Name::new_inline("b"),
                        predicate: Predicate::Boolean(true),
                    },
                );
                if !entailment(fold.context.clone(), constraint) {
                    panic!("if condition not a valid boolean")
                }

                let (fold, constraint) = fold.check_type(*then_branch, constraint_type.clone());
                let c1 = implication_constraint(
                    fresh_var.clone(),
                    RefinedBase {
                        base: Type::Scalar(ScalarType::Boolean),
                        binder: Name::new_inline("branch_"),
                        predicate: Predicate::Variable(fold.term_as_name(*condition)),
                    },
                    Some(constraint),
                );
                let (fold, constraint) = fold.check_type(*else_branch, constraint_type);
                let c2 = implication_constraint(
                    fresh_var,
                    RefinedBase {
                        base: Type::Scalar(ScalarType::Boolean),
                        binder: Name::new_inline("branch_"),
                        predicate: Predicate::Unary(
                            UnaryOperator::Negation,
                            Predicate::Variable(fold.term_as_name(*condition)).into(),
                        ),
                    },
                    Some(constraint),
                );
                (fold, Constraint::Conjunction(c1.into(), c2.into()))
            }
            _ => {
                let (fold, t, c) = self.synth_type(term_id);
                let (fold, c2) = fold.subtype(t.as_refined_base(), constraint_type);
                (fold, Constraint::make_conjunction(c, c2))
            }
        }
    }

    fn synth_type(self, term_id: TermId) -> (Self, RefinedType, Option<Constraint>) {
        let term = &self.body.expressions[term_id];
        match term {
            Term::Block {
```

```
                statements,
                trailing_expression,
            } => {
                let (fold, constraints) =
                    statements
                        .iter()
                        .fold(
                            (self, vec![]),
                            |(fold, mut constraints), statement| match statement {
                                Statement::Let(pattern, init_term) => {
                                  let (mut fold, ty, c) = fold.synth_type(*init_term);
                                    let pattern = &fold.body.patterns[*pattern];
                                    let (path, name) = match pattern {
                                        Pattern::Bind(name) => (
                                            Path {
                                                segments: vec![name.clone()],
                                            },
                                            name.clone(),
                                        ),
                                        _ => todo!(),
                                    };
                                    fold.context.bindings.push((path, ty.clone()));
                                    constraints.push((name, ty, c));
                                    (fold, constraints)
                                }
                                Statement::Expression(_) => todo!(),
                            },
                        );
                let (fold, base, c) = fold.synth_type(*trailing_expression);
                let constraint = constraints.into_iter().rfold(
                    c,
                    |inner_constraint, (name, ty, outer_constraint)| {
                        let implication_constraint =
                            implication_constraint(name, ty.as_refined_base(),
                            ↪   inner_constraint);
                        Some(Constraint::make_conjunction(
                            outer_constraint,
                            implication_constraint,
                        ))
                    },
                );
                (fold, base, constraint)
            }
            Term::Path(path) => match self
                .context
                .get(path)
                .expect(format!("{path:?} not in context.").as_str())
                .clone()
```

```
            {
                RefinedType::Base(RefinedBase {
                    base,
                    binder,
                    predicate,
                }) => {
                    assert!(binder != path.as_name());

                    let path_type = RefinedBase {
                        base,
                        binder: binder.clone(),
                        predicate: Predicate::Binary(
                            BinaryOperator::Logic(LogicOperator::And),
                            predicate.into(),
                            Predicate::Binary(
                                BinaryOperator::Compare(CompareOperator::Equality {
                                    negated: false,
                                }),
                                Predicate::Variable(binder).into(),
                                Predicate::Variable(path.as_name()).into(),
                            )
                            .into(),
                        ),
                    };
                    (self, path_type.into(), None)
                }
                dependent_fn => (self, dependent_fn, None),
            },
            Term::Literal(lit) => match lit {
                Literal::Integer(value, sufix) => {
                    (self, primitive_integer(*value, *sufix).into(), None)
                }
                Literal::Bool(value) => (self, primitive_bool(*value).into(), None),
            },
            Term::Unary(op, term) => self.synth_unary_term(op, *term),
           Term::Binary(op, lhs, rhs) => self.synth_binary_term(op, *lhs, *rhs).into(),
            Term::Call { callee, arguments } => {
                self.synth_call_term(*callee, arguments.clone()).into()
            }
            term => panic!("Unhandled term {:?}", term),
        }
    }

    pub(crate) fn synth_unary_term(
        self,
        op: &'a TermUnaryOp,
        term: Idx<Term>,
    ) -> (Fold, RefinedType, Option<Constraint>) {
```

```
        match op {
            TermUnaryOp::Minus => {
                let minus_signature = DependentFunction {
                    parameter: (
                        Name::new_inline("param0"),
                        RefinedBase {
                            base: Type::Scalar(ScalarType::Integer(IntegerKind::I64)),
                            binder: Name::new_inline("param0"),
                            predicate: Predicate::Boolean(true),
                        },
                    ),
                    tail_type: RefinedType::Base(RefinedBase {
                        base: Type::Scalar(ScalarType::Integer(IntegerKind::I64)),
                        binder: Name::new_inline("ret"),
                        predicate: Predicate::Binary(
                            BinaryOperator::Compare(CompareOperator::Equality {
                            ↪   negated: false }),
                            Predicate::Variable(Name::new_inline("ret")).into(),
                            Predicate::Unary(
                                UnaryOperator::Minus,
                               Predicate::Variable(Name::new_inline("param0")).into(),
                            )
                            .into(),
                        ),
                    })
                    .into(),
                };

                let (fold, ty, constraint) =
                    self.synth_function_call(minus_signature, vec![term], None);
                (fold, ty, constraint)
            }
            TermUnaryOp::Negation => {
                let minus_signature = DependentFunction {
                    parameter: (
                        Name::new_inline("param0"),
                        RefinedBase {
                            base: Type::Scalar(ScalarType::Boolean),
                            binder: Name::new_inline("param0"),
                            predicate: Predicate::Boolean(true),
                        },
                    ),
                    tail_type: RefinedType::Base(RefinedBase {
                        base: Type::Scalar(ScalarType::Boolean),
                        binder: Name::new_inline("ret"),
                        predicate: Predicate::Binary(
                            BinaryOperator::Compare(CompareOperator::Equality {
                            ↪   negated: false }),
```

```
                            Predicate::Variable(Name::new_inline("ret")).into(),
                            Predicate::Unary(
                                UnaryOperator::Negation,
                               Predicate::Variable(Name::new_inline("param0")).into(),
                            )
                            .into(),
                        ),
                    })
                    .into(),
                };

                let (fold, ty, constraint) =
                    self.synth_function_call(minus_signature, vec![term], None);
                (fold, ty, constraint)
            }
            TermUnaryOp::Reference => todo!(),
            TermUnaryOp::Dereference => todo!(),
        }
    }

    fn synth_binary_term(
        self,
        op: &'a BinaryOperator,
        lhs: Idx<Term>,
        rhs: Idx<Term>,
    ) -> (Fold, RefinedType, Option<Constraint>) {
        let lhs_ty = self.inference.type_of_expression[lhs].clone();
        let rhs_ty = self.inference.type_of_expression[rhs].clone();
        let sum_signature = DependentFunction {
            parameter: (
                Name::new_inline("param0"),
                RefinedBase {
                    base: lhs_ty.clone(),
                    binder: Name::new_inline("param0"),
                    predicate: Predicate::Boolean(true),
                },
            ),
            tail_type: RefinedType::Fn(DependentFunction {
                parameter: (
                    Name::new_inline("param1"),
                    RefinedBase {
                        base: rhs_ty,
                        binder: Name::new_inline("param1"),
                        predicate: Predicate::Boolean(true),
                    },
                ),
                tail_type: RefinedType::Base(RefinedBase {
                    base: match op {
```

```
                        BinaryOperator::Arithmetic(_) => lhs_ty,
                        BinaryOperator::Logic(_) | BinaryOperator::Compare(_) => {
                            Type::Scalar(ScalarType::Boolean)
                        }
                    },
                    binder: Name::new_inline("ret"),
                    predicate: Predicate::Binary(
                        BinaryOperator::Compare(CompareOperator::Equality { negated:
                        ↪  false }),
                        Predicate::Variable(Name::new_inline("ret")).into(),
                        Predicate::Binary(
                            *op,
                            Predicate::Variable(Name::new_inline("param0")).into(),
                            Predicate::Variable(Name::new_inline("param1")).into(),
                        )
                        .into(),
                    ),
                })
                .into(),
            })
            .into(),
        };

        let (fold, ty, constraint) = self.synth_function_call(sum_signature, vec![lhs,
        ↪  rhs], None);
        (fold, ty, constraint)
    }

    fn synth_call_term(
        self,
        callee: Idx<Term>,
        arguments: Vec<Idx<Term>>,
    ) -> (Self, RefinedType, Option<Constraint>) {
        let (fold, function_ty, constraint) = self.synth_type(callee);

        let (fold, ret_ty, constraint) =
            fold.synth_function_call(function_ty.as_dependent_function(), arguments,
            ↪  constraint);
        (fold, ret_ty, constraint)
    }

    fn synth_function_call(
        self,
        function_ty: DependentFunction,
        mut argument_terms: Vec<TermId>,
        constraint: Option<Constraint>,
    ) -> (Self, RefinedType, Option<Constraint>) {
        if argument_terms.is_empty() {
```

```
            (self, function_ty.into(), constraint)
        } else {
            let argument = argument_terms.pop().unwrap();
            // synth
            let (fold, ty, constraint) =
                self.synth_function_call(function_ty, argument_terms, constraint);

            let function_ty = ty.as_dependent_function();
            let (parameter_name, argument_type) = function_ty.parameter;

            // check
            let (fold, c) = fold.check_type(argument, argument_type.clone());

            let argument_name = fold.term_as_name(argument);

            // substitue
            let return_type =
                substitution_in_refined_type(*function_ty.tail_type, parameter_name,
                ↪   argument_name);
            (
                fold,
                return_type.into(),
                Some(Constraint::make_conjunction(constraint, c)),
            )
        }
    }

    fn make_fresh_var(&mut self) -> Name {
        let fresh = self.fresh_var_counter;
        self.fresh_var_counter += 1;
        Name {
            id: format!("__fresh{}", fresh).into(),
        }
    }

    fn term_as_name(&self, term_id: TermId) -> Name {
        match &self.body.expressions[term_id] {
            Term::Path(path) => path.as_name(),
            _ => panic!("not a path term"),
        }
    }
}

fn substitution_in_refined_base(ty: RefinedBase, old_name: Name, new_name: Name) ->
↪   RefinedBase {
    let RefinedBase {
        base,
        binder,
```

```
        predicate,
    } = ty;
    if new_name == binder {
        let new_binder = Name::new_inline(format!("{}{}", binder.id.as_str(),
        ↪  1).as_str());
        substitution_in_refined_base(
            RefinedBase {
                base,
                binder: new_binder.clone(),
                predicate: substitution(binder, new_binder, predicate),
            },
            old_name,
            new_name,
        )
    } else if old_name == binder {
        RefinedBase {
            base,
            binder,
            predicate,
        }
    } else {
        RefinedBase {
            base,
            binder,
            predicate: substitution(old_name, new_name, predicate),
        }
    }
}

fn substitution_in_function_type(
    function_ty: DependentFunction,
    old_name: Name,
    new_name: Name,
) -> DependentFunction {
    let DependentFunction {
        parameter: (arg_name, arg_type),
        tail_type,
    } = function_ty;
    if new_name == arg_name {
        let new_arg_name = Name::new_inline(format!("{}{}", arg_name.id.as_str(),
        ↪  1).as_str());
        substitution_in_function_type(
            DependentFunction {
                parameter: (
                    new_arg_name.clone(),
                    substitution_in_refined_base(arg_type, arg_name, new_arg_name),
                ),
                tail_type,
```

```
            },
            old_name,
            new_name,
        )
    } else if old_name == arg_name {
        DependentFunction {
            parameter: (
                arg_name,
                substitution_in_refined_base(arg_type, old_name, new_name),
            ),
            tail_type,
        }
    } else {
        DependentFunction {
            parameter: (
                arg_name,
                substitution_in_refined_base(arg_type, old_name.clone(),
                ↪   new_name.clone()),
            ),
            tail_type: substitution_in_refined_type(*tail_type, old_name,
            ↪   new_name).into(),
        }
    }
}

fn substitution_in_refined_type(ty: RefinedType, old_name: Name, new_name: Name) ->
↪   RefinedType {
    match ty {
        RefinedType::Base(base) => substitution_in_refined_base(base, old_name,
        ↪   new_name).into(),
        RefinedType::Fn(func) => substitution_in_function_type(func, old_name,
        ↪   new_name).into(),
    }
}

fn primitive_bool(value: bool) -> RefinedBase {
    let binder = Name::new_inline("lit");
    RefinedBase {
        base: Type::Scalar(ScalarType::Boolean),
        binder: binder.clone(),
        predicate: match value {
            true => Predicate::Variable(binder),
            false => Predicate::Unary(UnaryOperator::Negation,
            ↪   Predicate::Variable(binder).into()),
        },
    }
}

```

```rust
fn primitive_integer(value: u128, sufix: Option<BuiltinInteger>) -> RefinedBase {
    let binder = Name::new_inline("lit");
    RefinedBase {
        base: Type::Scalar(ScalarType::Integer(
            sufix.map_or(IntegerKind::I64, Into::into),
        )),
        binder: binder.clone(),
        predicate: Predicate::Binary(
            BinaryOperator::Compare(CompareOperator::Equality { negated: false }),
            Predicate::Variable(binder).into(),
            Predicate::Integer(value).into(),
        ),
    }
}

fn substitution(old: Name, new: Name, predicate: Predicate) -> Predicate {
    match predicate.clone() {
        Predicate::Variable(name) => {
            if name == old {
                Predicate::Variable(new)
            } else {
                predicate
            }
        }
        Predicate::Binary(op, lhs, rhs) => Predicate::Binary(
            op,
            substitution(old.clone(), new.clone(), *lhs).into(),
            substitution(old, new, *rhs).into(),
        ),
        Predicate::Unary(op, predicate) => {
            Predicate::Unary(op, substitution(old, new, *predicate).into())
        }
        Predicate::Boolean(_) | Predicate::Integer(_) => predicate,
    }
}

fn implication_constraint(
    name: Name,
    ty: RefinedBase,
    constraint: Option<Constraint>,
) -> Constraint {
    let RefinedBase {
        base,
        binder,
        predicate,
    } = ty;

    let constraint = Constraint::Implication {
```

```
        binder: name.clone(),
        base,
        antecedent: substitution(binder, name, predicate),
        consequent: constraint
            .unwrap_or(Constraint::Predicate(Predicate::Boolean(true)))
            .into(),
    };

    constraint
}
```

A.2 MODULE CODEGEN

```
use std::{
    cell::RefCell,
    collections::{BTreeMap, HashMap},
    fmt::Display,
    iter::FromIterator,
    sync::Arc,
};

use by_address::ByAddress;
use inkwell::{
    attributes::{Attribute, AttributeLoc},
    basic_block::BasicBlock,
    builder::Builder,
    context::Context,
    module::Module,
    targets::{
        CodeModel, InitializationConfig, RelocMode, Target, TargetData, TargetMachine, TargetTriple,
    },
    types::{AnyType, BasicType, BasicTypeEnum, FunctionType, IntType, StructType},
    values::{BasicValue, BasicValueEnum, FunctionValue, PointerValue},
    AddressSpace, IntPredicate, OptimizationLevel,
};

use hir::{
    check::type_inference::InferenceResult,
    check::{IntegerKind, ScalarType, Type},
    semantic_ir::{
        definition_map::{CallableDefinitionId, ValueConstructorId},
        name::Name,
        path_resolver::{Resolver, ValueNamespaceItem},
    },
    semantic_ir::{
        definition_map::{FunctionDefinitionId, TypeDefinitionId},
```

```
        path::Path,
        term::{
            ArithmeticOperator, BinaryOperator, Body, CompareOperator, Literal,
            ↪  LogicOperator,
            Ordering, Pattern, PatternId, Statement, Term, TermId, UnaryOperator,
        },
    },
    HirDatabase, SourceDatabase, Upcast,
};

#[salsa::query_group(CodeGenDatabaseStorage)]
pub trait CodeGenDatabase: HirDatabase + Upcast<dyn HirDatabase> {
    #[salsa::input]
    fn target(&self) -> CodeGenTarget;

    #[salsa::input]
    fn liquid(&self) -> bool;

    fn target_machine(&self) -> ByAddress<Arc<TargetMachine>>;

    fn build_assembly_ir(&self) -> ();
}

fn target_machine(db: &dyn CodeGenDatabase) -> ByAddress<Arc<TargetMachine>> {
    let target = db.target();

    match target.triple.arch {
        CodeGenTargetArch::X86_64 => {
            Target::initialize_x86(&InitializationConfig::default());
        }
    };

    let target_triple = TargetTriple::create(&target.triple.to_string());
    let target = Target::from_triple(&target_triple).unwrap_or_else(|err| {
        panic!(
            "Could not create LLVM target from target triple {}: {err:?}",
            target.triple
        )
    });
    let opt = OptimizationLevel::None;
    let reloc = RelocMode::Default;
    let model = CodeModel::Default;
    let cpu = "";
    let features = "";
    let target_machine = target
        .create_target_machine(&target_triple, cpu, features, opt, reloc, model)
        .unwrap();

```

```
    ByAddress(Arc::new(target_machine))
}

#[salsa::database(
    hir::SourceDatabaseStorage,
    hir::InternerStorage,
    hir::DefinitionsDatabaseStorage,
    hir::HirDatabaseStorage,
    CodeGenDatabaseStorage
)]
#[derive(Default)]
pub struct Database {
    storage: salsa::Storage<Self>,
}

impl Upcast<dyn hir::SourceDatabase> for Database {
    fn upcast(&self) -> &(dyn hir::SourceDatabase + 'static) {
        &*self
    }
}

impl Upcast<dyn hir::Interner> for Database {
    fn upcast(&self) -> &(dyn hir::Interner + 'static) {
        &*self
    }
}

impl Upcast<dyn hir::DefinitionsDatabase> for Database {
    fn upcast(&self) -> &(dyn hir::DefinitionsDatabase + 'static) {
        &*self
    }
}

impl Upcast<dyn hir::HirDatabase> for Database {
    fn upcast(&self) -> &(dyn hir::HirDatabase + 'static) {
        &*self
    }
}

impl salsa::Database for Database {}

#[derive(Clone)]
struct LocalBindingStack<'ink> {
    stack: Vec<Scope<'ink>>,
}

impl<'ink> LocalBindingStack<'ink> {
    pub fn get(&self, name: &Name) -> Option<Value<'ink>> {
```

```
        self.stack.iter().rev().find_map(|scope| scope.get(name))
    }

    pub fn push(&mut self, scope: Scope<'ink>) {
        self.stack.push(scope);
    }

    pub fn pop(&mut self) {
        self.stack.pop();
    }
}

impl<'ink> FromIterator<Scope<'ink>> for LocalBindingStack<'ink> {
    fn from_iter<I>(iter: I) -> Self
    where
        I: IntoIterator<Item = Scope<'ink>>,
    {
        let stack = iter.into_iter().collect();
        Self { stack }
    }
}

#[derive(Clone)]
struct Scope<'ink> {
    scope: Vec<Binding<'ink>>,
}

impl<'ink> Scope<'ink> {
    pub fn get(&self, name: &Name) -> Option<Value<'ink>> {
        self.scope
            .iter()
            .find_map(|binding| match &binding.name == name {
                true => Some(binding.value.clone()),
                false => None,
            })
    }
}

impl<'ink> FromIterator<Binding<'ink>> for Scope<'ink> {
    fn from_iter<I>(iter: I) -> Self
    where
        I: IntoIterator<Item = Binding<'ink>>,
    {
        let scope = iter.into_iter().collect();
        Self { scope }
    }
}

```

```
#[derive(Clone)]
struct Binding<'ink> {
    pub name: Name,
    pub value: Value<'ink>,
}

#[derive(Debug, Clone)]
struct Value<'a> {
    pub kind: ValueKind,
    pub value: BasicValueEnum<'a>,
}

impl<'a> Value<'a> {
    fn new(kind: ValueKind, value: BasicValueEnum<'a>) -> Self {
        Self { kind, value }
    }
}

#[derive(Debug, Clone, Copy)]
pub enum CodeGenTargetArch {
    X86_64,
}

impl Display for CodeGenTargetArch {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let string = match self {
            Self::X86_64 => "x86_64",
        };
        f.write_str(string)
    }
}

#[derive(Debug, Clone, Copy)]
pub enum CodeGenTargetVendor {
    PC,
}

impl Display for CodeGenTargetVendor {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let string = match self {
            Self::PC => "pc",
        };
        f.write_str(string)
    }
}

#[derive(Debug, Clone, Copy)]
pub enum CodeGenTargetSystem {
```

```
    Linux,
}

impl Display for CodeGenTargetSystem {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let string = match self {
            Self::Linux => "linux",
        };
        f.write_str(string)
    }
}

#[derive(Debug, Clone, Copy)]
pub enum CodeGenTargetABI {
    GNU,
}

impl Display for CodeGenTargetABI {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let string = match self {
            Self::GNU => "gnu",
        };
        f.write_str(string)
    }
}

#[derive(Debug, Clone, Copy)]
pub struct CodeGenTargetTriple {
    pub arch: CodeGenTargetArch,
    pub vendor: CodeGenTargetVendor,
    pub system: CodeGenTargetSystem,
    pub abi: CodeGenTargetABI,
}

impl Display for CodeGenTargetTriple {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.write_fmt(format_args!(
            "{}-{}-{}-{}",
            self.arch, self.vendor, self.system, self.abi
        ))
    }
}

#[derive(Debug, Clone, Copy)]
pub struct CodeGenTarget {
    pub triple: CodeGenTargetTriple,
}

```

```
pub struct CodeGenTypeCache<'db, 'context> {
    db: &'db dyn CodeGenDatabase,
    context: &'context Context,
    target_data: TargetData,
    adt_map: RefCell<HashMap<TypeDefinitionId, (StructType<'context>,
    ↪ TypeInfo<'context>)>>,
    adt_variant_map: RefCell<HashMap<ValueConstructorId, StructType<'context>>>,
}

impl<'db, 'context> CodeGenTypeCache<'db, 'context> {
    pub(crate) fn new(db: &'db dyn CodeGenDatabase, context: &'context Context) -> Self
    ↪ {
        Self {
            db,
            context,
            target_data: db.target_machine().get_target_data(),
            adt_map: RefCell::new(HashMap::new()),
            adt_variant_map: RefCell::new(HashMap::new()),
        }
    }

    fn llvm_type(&self, ty: &Type) -> BasicTypeEnum<'context> {
        match ty {
            Type::AbstractDataType(ty_loc_id) =>
            ↪ self.adt_struct_type(*ty_loc_id).into(),
            Type::FunctionDefinition(_) => todo!(),
            Type::Scalar(scalar) => self.scalar_type(scalar),
            Type::Pointer(inner) => {
                let inner = self.llvm_type(inner);
                let ptr_type = inner.ptr_type(AddressSpace::Generic);
                ptr_type.as_basic_type_enum()
            }
        }
    }

    fn type_info(&self, ty: &Type) -> TypeInfo<'context> {
        match ty {
            Type::AbstractDataType(ty_loc_id) => self.adt_struct_info(*ty_loc_id),
            Type::FunctionDefinition(_) => todo!(),
            Type::Scalar(_) => todo!(),
            Type::Pointer(_) => todo!(),
        }
    }

    fn bit_size(&self, ty: &Type) -> usize {
        let llvm_type = self.llvm_type(ty);
        self.target_data.get_bit_size(&llvm_type.as_any_type_enum()) as usize
    }
```

```

    fn target_data(&self) -> &TargetData {
        &self.target_data
    }

    fn adt_struct_info(&self, ty_loc_id: TypeDefinitionId) -> TypeInfo<'context> {
        self.adt_struct(ty_loc_id).1
    }

    fn adt_struct_type(&self, ty_loc_id: TypeDefinitionId) -> StructType<'context> {
        self.adt_struct(ty_loc_id).0
    }

    fn adt_struct(
        &self,
        ty_loc_id: TypeDefinitionId,
    ) -> (StructType<'context>, TypeInfo<'context>) {
        if let Some(value) = self.adt_map.borrow().get(&ty_loc_id) {
            return value.clone();
        };
        let ty_data = self.db.type_definition_data(ty_loc_id);
        let opaque = self.context.opaque_struct_type(ty_data.name.id.as_str());
        self.adt_map
            .borrow_mut()
            .insert(ty_loc_id, (opaque, TypeInfo::default()));

        let tag_bit_len = {
            let variant_count = ty_data.value_constructors.len();
            let mut bit_len = 0;
            while 1 << bit_len < variant_count {
                bit_len += 1;
            }
            bit_len
        };

        let tag_type = match tag_bit_len {
            0 => Some(self.context.i64_type()),
            1 => Some(self.context.i64_type()),
            2..=8 => Some(self.context.i64_type()),
            9..=16 => Some(self.context.i64_type()),
            17..=32 => Some(self.context.i64_type()),
            33..=64 => Some(self.context.i64_type()),
            _ => panic!("Tag for ADT too big! {tag_bit_len:?}"),
        };

        let value_constructor_ids = ty_data
            .value_constructors
            .iter()
```

```
            .map(|(id, _)| ValueConstructorId {
                type_definition_id: ty_loc_id,
                id,
            })
            .collect::<Vec<_>>();

        let ty_constructors = value_constructor_ids
            .iter()
            .cloned()
            .map(|id| self.value_constructor_struct(id));

        let bigest = ty_constructors
            .max_by_key(|struct_type| {
                self.target_data
                    .get_abi_size(&struct_type.as_any_type_enum())
            })
            .expect("Failed to get memsize of type variant");

        let field_types = tag_type
            .into_iter()
            .map(Into::into)
            .chain(std::iter::once(bigest.into()))
            .collect::<Vec<_>>();

        opaque.set_body(field_types.as_slice(), false);
        let tag_map = value_constructor_ids
            .into_iter()
            .map(|id| (id, u32::from(id.id.into_raw()) as usize))
            .collect();
        let adt_info = TypeInfo {
            tag: tag_type,
            tag_map,
        };
        let value = (opaque, adt_info);
        self.adt_map.borrow_mut().insert(ty_loc_id, value.clone());
        value
    }

    fn value_constructor_struct(&self, constructor_id: ValueConstructorId) -> StructType<'context> {
        if let Some(struct_type) = self.adt_variant_map.borrow().get(&constructor_id) {
            return *struct_type;
        }

        let constructor = self.db.callable_definition_signature(constructor_id.into());
        let struct_data = constructor
            .parameter_types
            .iter()
```

```
            .map(|ty| self.llvm_type(ty))
            .collect::<Vec<_>>();

        let ty_data = self
            .db
            .type_definition_data(constructor_id.type_definition_id);
        let variant_data = ty_data.value_constructor(constructor_id.id);

        let opaque = self
            .context
            .opaque_struct_type(format!("{}::{}", ty_data.name.id,
            ↪  variant_data.name.id).as_str());

        opaque.set_body(&struct_data, false);

        self.adt_variant_map
            .borrow_mut()
            .insert(constructor_id, opaque);
        opaque
    }

    fn scalar_type(&self, scalar: &ScalarType) -> BasicTypeEnum<'context> {
        match scalar {
            ScalarType::Integer(int_kind) => match int_kind {
                IntegerKind::I32 => self.context.i32_type().into(),
                IntegerKind::I64 => self.context.i64_type().into(),
            },
            ScalarType::Boolean => self.context.bool_type().into(),
        }
    }
}

#[derive(Default, Clone)]
struct TypeInfo<'context> {
    pub tag: Option<IntType<'context>>,
    pub tag_map: HashMap<ValueConstructorId, usize>,
}

struct CodeGenFunctionInfoCache<'db, 'context, 'type_cache> {
    db: &'db dyn CodeGenDatabase,
    context: &'context Context,
    type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
}

impl<'db, 'context, 'type_cache> CodeGenFunctionInfoCache<'db, 'context, 'type_cache>
↪  {
    pub(crate) fn new(
        db: &'db dyn CodeGenDatabase,
```

```
        context: &'context Context,
        type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
    ) -> Self {
        Self {
            db,
            context,
            type_cache,
        }
    }

    pub(crate) fn llvm_function_type(
        &self,
        function_id: &FunctionDefinitionId,
    ) -> FunctionType<'context> {
        let function_info = self.function_info(function_id);
        let function_signature =
        ↪  self.db.callable_definition_signature((*function_id).into());
        let parameter_types = function_signature
            .parameter_types
            .iter()
            .zip(function_info.parameter_kinds)
            .map(|(param_type, param_kind)| {
                let llvm_param_type = self.type_cache.llvm_type(param_type);
                match param_kind {
                    ValueKind::Indirect => llvm_param_type
                        .ptr_type(AddressSpace::Generic)
                        .as_basic_type_enum(),
                    ValueKind::Direct => llvm_param_type,
                }
            });
        let return_type = {
            let return_type = function_signature.return_type;
            self.type_cache.llvm_type(&return_type)
        };
        match function_info.return_kind {
            ValueKind::Indirect => {
                let return_type = return_type
                    .ptr_type(AddressSpace::Generic)
                    .as_basic_type_enum();
                let parameter_types =
                ↪  std::iter::once(return_type).chain(parameter_types);
                self.context.void_type().fn_type(
                    parameter_types
                        .map(Into::into)
                        .collect::<Vec<_>>()
                        .as_slice(),
                    false,
                )
```

```
            }
            ValueKind::Direct => return_type.fn_type(
                &parameter_types
                    .into_iter()
                    .map(Into::into)
                    .collect::<Vec<_>>(),
                false,
            ),
        }
    }

    pub(crate) fn function_info(&self, function_id: &FunctionDefinitionId) ->
    ↪   FunctionInfo {
        let free_integer_registers = 6;
        let function_signature =
        ↪   self.db.callable_definition_signature((*function_id).into());
        let return_bitsize = self.type_cache.bit_size(&function_signature.return_type);
        let (free_integer_registers, return_kind) = match return_bitsize {
            0..=128 => (free_integer_registers, ValueKind::Direct),
            _ => (free_integer_registers - 1, ValueKind::Indirect),
        };

        let (_, parameter_kinds) = function_signature.parameter_types.iter().fold(
            (free_integer_registers, Vec::new()),
            |(free_integer_registers, mut param_kinds), param_type| match
            ↪   free_integer_registers {
                0 => {
                    let bitsize = self.type_cache.bit_size(param_type);
                    if bitsize <= 64 {
                        param_kinds.push(ValueKind::Direct)
                    } else {
                        param_kinds.push(ValueKind::Indirect);
                    }
                    (free_integer_registers, param_kinds)
                }
                _ => {
                    let param_size = self.type_cache.bit_size(param_type);
                    let (free_integer_registers, param_kind) = match param_size {
                        1..=64 => (free_integer_registers - 1, ValueKind::Direct),
                        65..=128 => (free_integer_registers - 2, ValueKind::Direct),
                        129.. => (free_integer_registers, ValueKind::Indirect),
                        x => panic!("parameter size {x} not supported"),
                    };
                    param_kinds.push(param_kind);
                    (free_integer_registers, param_kinds)
                }
            },
        );
```

```
        FunctionInfo {
            return_kind,
            parameter_kinds,
        }
    }
}

struct FunctionInfo {
    pub return_kind: ValueKind,
    pub parameter_kinds: Vec<ValueKind>,
}

impl FunctionInfo {
    pub(crate) fn skip_return_param<T>(
        &self,
        iter: impl Iterator<Item = T>,
    ) -> impl Iterator<Item = T> {
        iter.skip(match self.return_kind {
            ValueKind::Indirect => 1,
            ValueKind::Direct => 0,
        })
    }

    pub(crate) fn get_return_type<'ctx>(
        &self,
        function_value: FunctionValue<'ctx>,
    ) -> BasicTypeEnum<'ctx> {
        match self.return_kind {
            ValueKind::Indirect => function_value.get_type().get_param_types()[0],
            ValueKind::Direct => function_value.get_type().get_return_type().unwrap(),
        }
    }
}

struct CodeGenFunctionValueCache<'db, 'context, 'module, 'type_cache,
↪   'function_info_cache> {
    db: &'db dyn CodeGenDatabase,
    context: &'context Context,
    module: &'module Module<'context>,
    type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
    function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<'db, 'context,
    ↪   'type_cache>,
    function_value_map: RefCell<HashMap<FunctionDefinitionId,
    ↪   FunctionValue<'context>>>,
}

impl<'db, 'context, 'module, 'type_cache, 'function_info_cache>
```

```
    CodeGenFunctionValueCache<'db, 'context, 'module, 'type_cache,
    ↪   'function_info_cache>
{
    pub(crate) fn new(
        db: &'db dyn CodeGenDatabase,
        context: &'context Context,
        module: &'module Module<'context>,
        type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
        function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<
            'db,
            'context,
            'type_cache,
        >,
    ) -> Self {
        Self {
            db,
            context,
            module,
            type_cache,
            function_info_cache,
            function_value_map: RefCell::new(HashMap::new()),
        }
    }

    fn llvm_function_value(&self, function_id: &FunctionDefinitionId) ->
    ↪   FunctionValue<'context> {
        if let Some(function_value) =
        ↪   self.function_value_map.borrow().get(function_id) {
            return *function_value;
        }

        let name = {
            let f_data = self.db.function_definition_data(*function_id);
            f_data.name.id
        };
        let llvm_function_type =
        ↪   self.function_info_cache.llvm_function_type(function_id);
        let function_value = self
            .module
            .add_function(name.as_str(), llvm_function_type, None);

        let function_info = self.function_info_cache.function_info(function_id);

        match function_info.return_kind {
            ValueKind::Indirect => {
                let ret_ty = llvm_function_type
                    .get_param_types()
                    .first()
```

```
                    .unwrap()
                    .into_pointer_type()
                    .get_element_type();
                let kind_id = Attribute::get_named_enum_kind_id("sret");
                let sret_attribute = self
                    .context
                    .create_type_attribute(kind_id, ret_ty.as_any_type_enum());
                let kind_id = Attribute::get_named_enum_kind_id("noalias");
                let noalias_attribute = self.context.create_enum_attribute(kind_id, 0);
                function_value.add_attribute(AttributeLoc::Param(0), sret_attribute);
                function_value.add_attribute(AttributeLoc::Param(0), noalias_attribute);
            }
            _ => (),
        };

        for ((param_index, param_type), param_kind) in function_info
            .skip_return_param(llvm_function_type.get_param_types().iter().enumerate())
            .zip(function_info.parameter_kinds)
        {
            if let ValueKind::Indirect = param_kind {
                let kind_id = Attribute::get_named_enum_kind_id("byval");
                let byval_attribute = self.context.create_type_attribute(
                    kind_id,
                    param_type
                        .into_pointer_type()
                        .get_element_type()
                        .as_any_type_enum(),
                );
                function_value.add_attribute(
                    AttributeLoc::Param(param_index.try_into().unwrap()),
                    byval_attribute,
                )
            }
        }
        self.function_value_map
            .borrow_mut()
            .insert(*function_id, function_value);
        function_value
    }
}

struct CodeGenFunctionBodyLoweringContext<
    'db,
    'context,
    'module,
    'type_cache,
    'function_info_cache,
    'function_value_cache,
```

```
> {
    db: &'db dyn CodeGenDatabase,
    context: &'context Context,
    module: &'module Module<'context>,
    type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
    function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<'db, 'context,
    ↪  'type_cache>,
    function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
        'db,
        'context,
        'module,
        'type_cache,
        'function_info_cache,
    >,
}

impl<'db, 'context, 'module, 'type_cache, 'function_info_cache, 'function_value_cache>
    CodeGenFunctionBodyLoweringContext<
        'db,
        'context,
        'module,
        'type_cache,
        'function_info_cache,
        'function_value_cache,
    >
{
    pub(crate) fn new(
        db: &'db dyn CodeGenDatabase,
        context: &'context Context,
        module: &'module Module<'context>,
        type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
        function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<
            'db,
            'context,
            'type_cache,
        >,
        function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
            'db,
            'context,
            'module,
            'type_cache,
            'function_info_cache,
        >,
    ) -> Self {
        Self {
            db,
            context,
            module,
```

```
            type_cache,
            function_info_cache,
            function_value_cache,
        }
    }

    pub(crate) fn build_function_body(&self, function_id: &FunctionDefinitionId) {
        let llvm_function_value =
        ↪   self.function_value_cache.llvm_function_value(function_id);
        let llvm_body = self.context.append_basic_block(llvm_function_value, "");
        let builder = self.context.create_builder();
        builder.position_at_end(llvm_body);
        let body = self.db.body_of_definition(*function_id);
        let function_signature =
        ↪   self.db.callable_definition_signature((*function_id).into());
        let function_info = self.function_info_cache.function_info(function_id);

        let return_value_ptr = match function_info.return_kind {
            ValueKind::Indirect => Some(
                llvm_function_value
                    .get_first_param()
                    .unwrap()
                    .into_pointer_value(),
            ),
            ValueKind::Direct => {
                let ret_type =
                ↪   llvm_function_value.get_type().get_return_type().unwrap();
                let bit_size = self.type_cache.target_data().get_bit_size(&ret_type);
                if bit_size <= 64 {
                    None
                } else {
                    Some(builder.build_alloca(ret_type, ""))
                }
            }
        };

        let parameter_scope = body
            .parameters
            .iter()
            .zip(
                function_info
                    .skip_return_param(llvm_function_value.get_param_iter())
                    .zip(function_info.parameter_kinds)
                    .zip(function_signature.parameter_types),
            )
            .map(|(pattern_id, ((parameter, parameter_kind), param_ty))| {
                let pattern = &body.patterns[*pattern_id];
                match pattern {
```

```
                    Pattern::Deconstructor(_, _) => todo!("Unsing unsuported path
                    ↪   pattern"),
                    Pattern::Bind(name) => {
                        parameter.set_name(&name.id);
                        let binding_value = match parameter_kind {
                            ValueKind::Indirect => Value {
                                kind: parameter_kind,
                                value: parameter,
                            },
                            ValueKind::Direct => {
                                if self.type_cache.bit_size(&param_ty) <= 64 {
                                    Value {
                                        kind: ValueKind::Direct,
                                        value: parameter,
                                    }
                                } else {
                                    let param_ptr =
                                        builder.build_alloca(parameter.get_type(),
                                        ↪   &name.id);
                                    builder.build_store(param_ptr, parameter);
                                    Value {
                                        kind: ValueKind::Indirect,
                                        value: param_ptr.into(),
                                    }
                                }
                            }
                        };
                        Binding {
                            name: name.clone(),
                            value: binding_value,
                        }
                    }
                }
            })
            .collect::<Scope>();

        let binding_stack = std::iter::once(parameter_scope).collect();

        let builder = self.context.create_builder();
        builder.position_at_end(llvm_body);

        let return_value = ExpressionLowerer::new(
            self.db,
            &self.context,
            &builder,
            &self.type_cache,
            &self.function_info_cache,
            &self.function_value_cache,
```

```
            *function_id,
            binding_stack,
        )
        .fold_expression(return_value_ptr, body.root_expression);

        let return_value = match return_value_ptr {
            None => {
                let Value { kind, value } = return_value.unwrap();
                let value = match kind {
                    ValueKind::Indirect => builder.build_load(value.into_pointer_value(), ""),
                    ValueKind::Direct => value,
                };
                Some(value)
            }
            Some(ptr) => match function_info.return_kind {
                ValueKind::Indirect => None,
                ValueKind::Direct => Some(builder.build_load(ptr, "")),
            },
        };

        builder.build_return(return_value.as_ref().map(|val| val as &dyn BasicValue));
    }
}

struct CodeGenFunctionArgumentInfo {
    kind: ArgumentKind,
}

enum ArgumentKind {
    Direct,
    Indirect,
}

#[derive(Clone, Copy)]
enum ReturnKind {
    ArgumentPointer,
    RegisterValue,
}

#[derive(Debug, Clone, Copy)]
enum ValueKind {
    Indirect,
    Direct,
}

struct FunctionIr<'ink> {
    pub value: FunctionValue<'ink>,
```

```
    pub return_kind: ReturnKind,
    pub parameters_kind: Vec<ValueKind>,
}

impl FunctionIr<'_> {
    fn parameters(&self) -> impl Iterator<Item = (BasicValueEnum, &ValueKind)> {
        match self.return_kind {
            ReturnKind::ArgumentPointer => self
                .value
                .get_param_iter()
                .zip(self.parameters_kind.iter())
                .skip(1),
            ReturnKind::RegisterValue => self
                .value
                .get_param_iter()
                .zip(self.parameters_kind.iter())
                .skip(0),
        }
    }
}

pub fn compile_text(source: String, target: CodeGenTarget, liquid: bool) {
    let mut db = Database::default();
    db.set_source_file_text(source);
    db.set_target(target);
    db.set_liquid(liquid);
    db.build_assembly_ir()
}

pub fn build_assembly_ir(db: &dyn CodeGenDatabase) {
    let context = Context::create();

    let target_machine = db.target_machine();
    let target_data = target_machine.get_target_data();

    let llvm_module = context.create_module("ekitai_module");
    llvm_module.set_data_layout(&target_data.get_data_layout());
    llvm_module.set_triple(&target_machine.get_triple());

    let type_cache = CodeGenTypeCache::new(db, &context);
   let function_info_cache = CodeGenFunctionInfoCache::new(db, &context, &type_cache);
    let function_value_cache = CodeGenFunctionValueCache::new(
        db,
        &context,
        &llvm_module,
        &type_cache,
        &function_info_cache,
    );
```

```rust

    let function_body_builder = CodeGenFunctionBodyLoweringContext::new(
        db,
        &context,
        &llvm_module,
        &type_cache,
        &function_info_cache,
        &function_value_cache,
    );

    let def_map = db.source_file_definitions_map();
    for function_id in def_map.root_module_item_scope().iter_function_locations() {
        if db.liquid() {
            if !hir::liquid::check_abstraction(db.upcast(), *function_id) {
                println!("Type error");
                return;
            }
        }
        function_body_builder.build_function_body(function_id)
    }

    println!("{}", llvm_module.print_to_string().to_string());
    if let Err(err) = llvm_module.verify() {
        let err = err.to_string();
        panic!("Could not verify llvm: {err}");
    };
    let _ = llvm_module.print_to_file("out.ll");
}

struct ExpressionLowerer<
    'db,
    'context,
    'module,
    'builder,
    'type_cache,
    'function_info_cache,
    'function_value_cache,
> {
    db: &'db dyn CodeGenDatabase,
    context: &'context Context,
    builder: &'builder Builder<'context>,
    type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
    function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<'db, 'context,
    ↪   'type_cache>,
    function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
        'db,
        'context,
        'module,
```

```
        'type_cache,
        'function_info_cache,
    >,
    function_id: FunctionDefinitionId,
    body: Body,
    binding_stack: RefCell<LocalBindingStack<'context>>,
    inference: InferenceResult,
}

impl<
        'db,
        'context,
        'module,
        'builder,
        'type_cache,
        'function_info_cache,
        'function_value_cache,
    >
    ExpressionLowerer<
        'db,
        'context,
        'module,
        'builder,
        'type_cache,
        'function_info_cache,
        'function_value_cache,
    >
{
    pub fn new(
        db: &'db dyn CodeGenDatabase,
        context: &'context Context,
        builder: &'builder Builder<'context>,
        type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
        function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<
            'db,
            'context,
            'type_cache,
        >,
        function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
            'db,
            'context,
            'module,
            'type_cache,
            'function_info_cache,
        >,
        function_id: FunctionDefinitionId,
        binding_stack: LocalBindingStack<'context>,
    ) -> Self {
```

```
        Self {
            db,
            context,
            builder,
            type_cache,
            function_info_cache,
            function_value_cache,
            function_id,
            body: db.body_of_definition(function_id),
            binding_stack: RefCell::new(binding_stack),
            inference: db.infer_body_expression_types(function_id),
        }
    }

    pub fn get_owener_function_value(&self) -> FunctionValue {
        self.function_value_cache
            .llvm_function_value(&self.function_id)
    }

    fn get_alloca_builder(&self) -> Builder<'context> {
        let builder = self.context.create_builder();
        let block = self
            .get_owener_function_value()
            .get_first_basic_block()
            .unwrap();
        match block.get_first_instruction() {
            Some(instruction) => builder.position_before(&instruction),
            None => builder.position_at_end(block),
        };
        builder
    }

    pub fn fold_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        expr_id: TermId,
    ) -> Option<Value<'context>> {
        let expr = &self.body.expressions[expr_id];
        match expr {
            Term::Block {
                statements,
                trailing_expression,
            } => {
                for statement in statements {
                    match statement {
                        Statement::Let(pattern_id, expr_id) => {
                            let pattern = &self.body.patterns[*pattern_id];
                            let value = self.fold_expression(None, *expr_id).unwrap();
```

```
                            let binding = match pattern {
                                Pattern::Deconstructor(_, _) => todo!(),
                                Pattern::Bind(name) => Binding {
                                    name: name.clone(),
                                    value,
                                },
                            };
                            self.binding_stack
                                .borrow_mut()
                                .push(std::iter::once(binding).collect());
                        }
                        Statement::Expression(expr_id) => {
                            self.fold_expression(None, *expr_id);
                        }
                    }
                }
               let value = self.fold_expression(indirect_value, *trailing_expression);
                for statement in statements {
                    if let Statement::Let(_, _) = statement {
                        self.binding_stack.borrow_mut().pop();
                    };
                }
                value
            }
            Term::If {
                condition,
                then_branch,
                else_branch,
            } => self.fold_if_expression(indirect_value, condition, then_branch,
            ↪   else_branch),
            Term::Match { matchee, case_list } => {
                let resolver =
                    Resolver::new_for_expression(self.db.upcast(), self.function_id,
                    ↪   expr_id);
                self.fold_match_expression(indirect_value, *matchee, case_list,
                ↪   resolver)
            }
            Term::Call { callee, arguments } => {
                self.fold_call_expression(indirect_value, callee, arguments)
            }
            Term::Binary(operator, lhs, rhs) => {
                self.fold_binary_expression(indirect_value, operator, lhs, rhs)
            }
            Term::Unary(op, expr) => self.fold_unary_expression(indirect_value, op,
            ↪   expr),
            Term::Path(path) => {
                let resolver =
```

```
                    Resolver::new_for_expression(self.db.upcast(), self.function_id,
                    ↪   expr_id);
                self.fold_path_expression(indirect_value, path, resolver)
            }
            Term::Literal(literal) => {
                self.fold_literal_expression(indirect_value, expr_id, literal)
            }
            Term::New(inner) => self.fold_new_expression(indirect_value, *inner),
        }
    }

    fn fold_if_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        condition: &TermId,
        then_branch: &TermId,
        else_branch: &TermId,
    ) -> Option<Value<'context>> {
        let comparison = self
            .fold_expression(None, *condition)
            .unwrap()
            .value
            .into_int_value();

        let then_block = self
            .context
            .append_basic_block(self.get_owener_function_value(), "then");
        let else_block = self
            .context
            .append_basic_block(self.get_owener_function_value(), "else");
        let merge_block = self
            .context
            .append_basic_block(self.get_owener_function_value(), "merge");

        self.builder
            .build_conditional_branch(comparison, then_block, else_block);

        self.builder.position_at_end(then_block);
        let then_value = self.fold_expression(indirect_value, *then_branch);
        let then_block = self.builder.get_insert_block().unwrap();
        self.builder.build_unconditional_branch(merge_block);

        self.builder.position_at_end(else_block);
        let else_value = self.fold_expression(indirect_value, *else_branch);
        let else_block = self.builder.get_insert_block().unwrap();
        self.builder.build_unconditional_branch(merge_block);

        self.builder.position_at_end(merge_block);
```

```rust
        match indirect_value {
            Some(ptr) => None,
            None => {
                let (
                    Value {
                        kind: then_kind,
                        value: then_value,
                    },
                    Value {
                        kind: else_kind,
                        value: else_value,
                    },
                ) = (then_value.unwrap(), else_value.unwrap());
                match (then_kind, else_kind) {
                    (ValueKind::Direct, ValueKind::Direct)
                    | (ValueKind::Indirect, ValueKind::Indirect) => {
                        let phi = self.builder.build_phi(then_value.get_type(), "phi");
                        phi.add_incoming(&[(&then_value, then_block), (&else_value, else_block)]);
                        Some(Value::new(then_kind, phi.as_basic_value()))
                    }
                    _ => panic!(),
                }
            }
        }
    }

    fn fold_match_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        matchee: TermId,
        case_list: &[(PatternId, TermId)],
        resolver: Resolver,
    ) -> Option<Value<'context>> {
        let Value {
            kind: matchee_kind,
            value: matchee_value,
        } = self.fold_expression(None, matchee).unwrap();

        let matchee_type = &self.inference.type_of_expression[matchee];
        let TypeInfo { tag, tag_map } = self.type_cache.type_info(matchee_type);

        let tag_value = match matchee_kind {
            ValueKind::Indirect => {
                let matchee_ptr = matchee_value.into_pointer_value();
                match tag {
                    Some(_) => {
```

```
                        let tag_ptr = self.builder.build_struct_gep(matchee_ptr, 0, "").unwrap();
                        Some(self.builder.build_load(tag_ptr, "").into_int_value())
                    }
                    None => None,
                }
            }
            ValueKind::Direct => match tag {
                Some(_) => Some(
                    self.builder
                        .build_extract_value(matchee_value.into_struct_value(), 0, "")
                        .unwrap()
                        .into_int_value(),
                ),
                None => None,
            },
        };

        match tag_value {
            Some(tag_value) => {
                let (patterns_per_tag_value, else_patterns) = case_list
                    .iter()
                    .cloned()
                    .map(|(pattern_id, expression_id)| {
                        let case_pattern = &self.body.patterns[pattern_id];
                        match case_pattern {
                            Pattern::Deconstructor(path, _) => {
                                let constructor_id = resolver
                                    .resolve_path_in_value_namespace(self.db.upcast(), path)
                                    .map(|item| match item {
                                       ValueNamespaceItem::ValueConstructor(id) => id,
                                        _ => panic!(),
                                    })
                                    .unwrap();
                                let tag_value = tag_map.get(&constructor_id).unwrap();
                                ((pattern_id, expression_id), Some(*tag_value))
                            }
                            Pattern::Bind(_) => ((pattern_id, expression_id), None),
                        }
                    })
                    .fold(
                        (BTreeMap::new(), Vec::new()),
                        |(mut tag_patterns_map, mut else_patterns), (case, tag)| match tag {
                            None => {
                                else_patterns.push(case);
                                (tag_patterns_map, else_patterns)
```

```
                    }
                    Some(tag) => {
                 tag_patterns_map.entry(tag).or_insert(vec![case]).push(case);
                        (tag_patterns_map, else_patterns)
                    }
                },
            );

        let else_block = self
            .context
           .append_basic_block(self.get_owener_function_value(), "case.else");
        let merge_block = self
            .context
          .append_basic_block(self.get_owener_function_value(), "case.merge");

        let (switch_cases, cases_and_blocks) = patterns_per_tag_value
            .into_iter()
            .map(|(tag, patterns)| {
                let tag_value = tag_value.get_type().const_int(tag as u64,
                ↪   false);
                let case_block = self
                    .context
                    .prepend_basic_block(merge_block, format!("br.{}.tag",
                    ↪   tag).as_str());
                ((tag_value, case_block), (case_block, patterns))
            })
            .unzip::<_, (BasicBlock, Vec<(PatternId, TermId)>), Vec<_>,
            ↪   Vec<_>>();

        self.builder
            .build_switch(tag_value, else_block,
            ↪   switch_cases.as_slice().as_ref());

        self.builder.position_at_end(else_block);
        self.builder.build_unreachable();

        let case_values = cases_and_blocks
            .into_iter()
            .map(|(case_block, cases)| {
                self.builder.position_at_end(case_block);
                let (pattern_id, expression_id) = cases.first().unwrap();
                let pattern = &self.body.patterns[*pattern_id];
                match pattern {
                    Pattern::Deconstructor(_, sub_patterns) => {
                        if sub_patterns.is_empty() {
                            let case_value =
                                self.fold_expression(indirect_value,
                                ↪   *expression_id);
```

```
 self.builder.build_unconditional_branch(merge_block);
    let case_block =
    ↪   self.builder.get_insert_block().unwrap();
    (case_value, case_block)
} else {
    let inner_value = match matchee_kind {
        ValueKind::Indirect => {
            let inner_ptr = self
                .builder
                .build_struct_gep(
                   matchee_value.into_pointer_value(),
                    1,
                    "",
                )
                .unwrap();
            Value::new(
                ValueKind::Indirect,
                inner_ptr.as_basic_value_enum(),
            )
        }
        ValueKind::Direct => {
            let inner_value = self
                .builder
                .build_extract_value(
                    matchee_value.into_struct_value(),
                    1,
                    "",
                )
                .unwrap();
            Value::new(ValueKind::Direct, inner_value)
        }
    };

    let scope = sub_patterns
        .iter()
        .enumerate()
        .map(|(index, sub_pattern_id)| {
            let sub_pattern =
            ↪   &self.body.patterns[*sub_pattern_id];
            match sub_pattern {
                Pattern::Bind(name) => match
                ↪   inner_value.kind {
                    ValueKind::Indirect => {
                        let bind_ptr = self
                            .builder
                            .build_struct_gep(
                                inner_value
                                    .value
```

```
            .into_pointer_value(),
            index as u32,
            "",
        )
        .unwrap();
    let value = if self
        .type_cache
        .target_data()
        .get_bit_size(
            &bind_ptr
                .get_type()
              .get_element_type(),
        )
        <= 64
    {
        Value::new(
            ValueKind::Direct,
            self.builder
        .build_load(bind_ptr, ""),
        )
    } else {
        Value::new(
            ValueKind::Indirect,
   bind_ptr.as_basic_value_enum(),
        )
    };
    Binding {
        name: name.clone(),
        value,
    }
}
ValueKind::Direct => {
    let value = self
        .builder
        .build_extract_value(
            inner_value
                .value
             .into_struct_value(),
            index as u32,
            "",
        )
        .unwrap();
    Binding {
        name: name.clone(),
        value: Value {
          kind: ValueKind::Direct,
            value,
        },
```

```
                                                }
                                            }
                                        },
                                      Pattern::Deconstructor(_, _) => todo!(),
                                    }
                                })
                                .collect();

                            self.binding_stack.borrow_mut().push(scope);
                            let case_value =
                                self.fold_expression(indirect_value,
                                ↪   *expression_id);
                         self.builder.build_unconditional_branch(merge_block);
                            self.binding_stack.borrow_mut().pop();
                            let case_block =
                            ↪   self.builder.get_insert_block().unwrap();
                            (case_value, case_block)
                        }
                    }
                    Pattern::Bind(_) => todo!(),
                }
            })
            .collect::<Vec<_>>();

        self.builder.position_at_end(merge_block);
        match indirect_value {
            Some(_) => None,
            None => {
                let first_value =
                ↪   case_values.first().unwrap().0.as_ref().unwrap();
                let phi = self.builder.build_phi(first_value.value.get_type(),
                ↪   "phi");
                phi.add_incoming(
                    case_values
                        .iter()
                        .map(|(case_value, case_block)| {
                            (
                                &case_value.as_ref().unwrap().value as &dyn
                                ↪   BasicValue,
                                *case_block,
                            )
                        })
                        .collect::<Vec<_>>()
                        .as_slice(),
                );

                Some(Value::new(first_value.kind, phi.as_basic_value()))
            }
```

```
                }
            }

            None => todo!(),
        }
    }

    fn fold_call_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        callee: &TermId,
        arguments: &[TermId],
    ) -> Option<Value<'context>> {
        let callee_type = &self.inference.type_of_expression[*callee];
        let callable_definition = match callee_type {
            Type::FunctionDefinition(callable) => callable,
            _ => panic!("call has no callable type."),
        };
        match callable_definition {
            CallableDefinitionId::FunctionDefinition(id) => {
                let function_info = self.function_info_cache.function_info(&id);
                let function_value = self.function_value_cache.llvm_function_value(&id);

                let return_type = function_info.get_return_type(function_value);

                let indirect_function_return = match function_info.return_kind {
                    ValueKind::Direct => None,
                    ValueKind::Indirect => match indirect_value {
                        Some(ptr) => Some(ptr.as_basic_value_enum()),
                        None => Some(
                            self.get_alloca_builder()
                                .build_alloca(
                                    BasicTypeEnum::try_from(
                                    return_type.into_pointer_type().get_element_type(),
                                    )
                                    .unwrap(),
                                    "",
                                )
                                .as_basic_value_enum(),
                        ),
                    },
                };

                let arguments = indirect_function_return.into_iter().chain(
                    arguments
                        .iter()
                        .zip(function_info.parameter_kinds.iter())
                        .map(|(argument_expr, parameter_kind)| {
```

```
                let Value {
                    kind: argument_kind,
                    value: argument_value,
                } = self.fold_expression(None, *argument_expr).unwrap();
              let argument_value = match (parameter_kind, argument_kind) {
                    (ValueKind::Indirect, ValueKind::Indirect)
                    | (ValueKind::Direct, ValueKind::Direct) =>
                    ↪   argument_value,
                    (ValueKind::Direct, ValueKind::Indirect) => self
                        .builder
                     .build_load(argument_value.into_pointer_value(), ""),
                    (ValueKind::Indirect, ValueKind::Direct) => {
                        let ptr = self
                            .get_alloca_builder()
                            .build_alloca(argument_value.get_type(), "");
                        self.builder.build_store(ptr, argument_value);
                        ptr.as_basic_value_enum()
                    }
                };
                argument_value
            }),
    );

    let call_value = self.builder.build_call(
        function_value,
        arguments.map(|x| x.into()).collect::<Vec<_>>().as_slice(),
        "",
    );

    match indirect_value {
        Some(_) => None,
        None => match function_info.return_kind {
            ValueKind::Indirect => Some(Value::new(
                ValueKind::Indirect,
                indirect_function_return.unwrap(),
            )),
            ValueKind::Direct => Some(Value::new(
                ValueKind::Direct,
                call_value.try_as_basic_value().unwrap_left(),
            )),
        },
    }
}
CallableDefinitionId::ValueConstructor(constructor_id) => {
    let (struct_type, type_info) = self
        .type_cache
        .adt_struct(constructor_id.type_definition_id);
    match indirect_value {
```

```
                    Some(ptr) => {
                        if let Some(tag_type) = type_info.tag {
                            let tag_value = type_info.tag_map[&constructor_id];
                          let tag_value = tag_type.const_int(tag_value as u64, false);
                            let tag_ptr = self.builder.build_struct_gep(ptr, 0,
                            ↪   "").unwrap();
                            self.builder.build_store(tag_ptr, tag_value);
                            let variant_ptr = self.builder.build_struct_gep(ptr, 1,
                            ↪   "").unwrap();
                            let variant_ptr = self
                                .builder
                                .build_bitcast(
                                    variant_ptr,
                                    self.type_cache
                                        .value_constructor_struct(*constructor_id)
                                        .ptr_type(AddressSpace::Generic),
                                    "",
                                )
                                .into_pointer_value();
                            let arguments = arguments
                                .iter()
                                .map(|argument| self.fold_expression(None,
                                ↪   *argument).unwrap());
                            for (index, argument) in arguments.enumerate() {
                                let argument_ptr = self
                                    .builder
                                    .build_struct_gep(variant_ptr, index as u32, "")
                                    .unwrap();
                                let value = match argument.kind {
                                    ValueKind::Indirect => {
                                        if !argument_ptr
                                            .get_type()
                                            .get_element_type()
                                            .is_pointer_type()
                                        {
                                            self.builder
                                      .build_load(argument.value.into_pointer_value(),
                                                ↪   "")
                                        } else {
                                            argument.value
                                        }
                                    }
                                    ValueKind::Direct => argument.value,
                                };
                                self.builder.build_store(argument_ptr, value);
                            }
                        } else {
                            todo!()
```

```
                        }
                        None
                    }
                    None => {
                        let type_size = self
                            .type_cache
                            .target_data()
                            .get_bit_size(&struct_type.as_any_type_enum());

                        let value = struct_type.const_zero();
                        let TypeInfo { tag, tag_map } = type_info;
                        let tag_value = tag.map(|tag_type| {
                            tag_type.const_int(tag_map[&constructor_id] as u64, true)
                        });
                        let value = if let Some(tag_value) = tag_value {
                            if type_size <= 64 {
                                let value = self
                                    .builder
                                    .build_insert_value(value, tag_value, 0, "")
                                    .unwrap()
                                    .as_basic_value_enum();
                                Value::new(ValueKind::Direct, value)
                            } else {
                          let ptr = self.get_alloca_builder().build_alloca(struct_type,
                                 ↪  "");
                                self.builder.build_store(
                                    ptr,
                                    ptr.get_type()
                                        .get_element_type()
                                        .into_struct_type()
                                        .const_zero(),
                                );
                                let tag_ptr = self.builder.build_struct_gep(ptr, 0,
                                 ↪  "").unwrap();
                                self.builder.build_store(tag_ptr, tag_value);
                                let variant_ptr =
                                   self.builder.build_struct_gep(ptr, 1, "").unwrap();
                                let variant_ptr = self
                                    .builder
                                    .build_bitcast(
                                        variant_ptr,
                                        self.type_cache
                                            .value_constructor_struct(*constructor_id)
                                            .ptr_type(AddressSpace::Generic),
                                        "",
                                    )
                                    .into_pointer_value();
                                let arguments = arguments
```

```
                                    .iter()
                                    .map(|argument| self.fold_expression(None,
                                    ↪  *argument).unwrap());
                                for (index, argument) in arguments.enumerate() {
                                    let argument_ptr = self
                                        .builder
                                      .build_struct_gep(variant_ptr, index as u32, "")
                                        .unwrap();
                                    let value = match argument.kind {
                                        ValueKind::Indirect => self
                                            .builder
                                      .build_load(argument.value.into_pointer_value(),
                                            ↪  ""),
                                        ValueKind::Direct => argument.value,
                                    };
                                    self.builder.build_store(argument_ptr, value);
                                }
                                Value::new(ValueKind::Indirect,
                                ↪  ptr.as_basic_value_enum())
                            }
                        } else {
                            todo!()
                        };
                        Some(value)
                    }
                }
            }
        }
    }

    fn fold_binary_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        operator: &BinaryOperator,
        lhs: &TermId,
        rhs: &TermId,
    ) -> Option<Value<'context>> {
        let lhs = self
            .fold_expression(None, *lhs)
            .unwrap()
            .value
            .into_int_value();
        let rhs = self
            .fold_expression(None, *rhs)
            .unwrap()
            .value
            .into_int_value();
        let int_value = match operator {
```

```
            BinaryOperator::Arithmetic(arithmetic_op) => match arithmetic_op {
                ArithmeticOperator::Add => self.builder.build_int_add(lhs, rhs, ""),
                ArithmeticOperator::Sub => self.builder.build_int_sub(lhs, rhs, ""),
                ArithmeticOperator::Div => self.builder.build_int_signed_div(lhs, rhs,
                ↪   ""),
                ArithmeticOperator::Mul => self.builder.build_int_mul(lhs, rhs, ""),
                ArithmeticOperator::Rem => self.builder.build_int_signed_rem(lhs, rhs,
                ↪   ""),
            },
            BinaryOperator::Compare(compare_op) => {
                let predicate = match compare_op {
                    CompareOperator::Equality { negated } => match negated {
                        true => IntPredicate::NE,
                        false => IntPredicate::EQ,
                    },
                    CompareOperator::Order { ordering, strict } => match (ordering,
                    ↪   strict) {
                        (Ordering::Less, true) => IntPredicate::SLT,
                        (Ordering::Less, false) => IntPredicate::SLE,
                        (Ordering::Greater, true) => IntPredicate::SGT,
                        (Ordering::Greater, false) => IntPredicate::SGE,
                    },
                };
                self.builder.build_int_compare(predicate, lhs, rhs, "")
            }
            BinaryOperator::Logic(logic_op) => match logic_op {
                LogicOperator::And => self.builder.build_and(lhs, rhs, ""),
                LogicOperator::Or => self.builder.build_or(lhs, rhs, ""),
            },
        };
        match indirect_value {
            Some(ptr) => {
                self.builder
                    .build_store(ptr, int_value.as_basic_value_enum());
                None
            }
            None => Some(Value::new(ValueKind::Direct, int_value.into())),
        }
    }

    fn fold_unary_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        operator: &UnaryOperator,
        expression: &TermId,
    ) -> Option<Value<'context>> {
        let Value { value: expr, kind } = self.fold_expression(None,
        ↪   *expression).unwrap();
```

```
        let (kind, value) = match operator {
            UnaryOperator::Minus => (
                kind,
                self.builder
                    .build_int_sub(
                        expr.get_type().into_int_type().const_zero(),
                        expr.into_int_value(),
                        "",
                    )
                    .into(),
            ),
            UnaryOperator::Negation => (
                kind,
                self.builder.build_not(expr.into_int_value(), "").into(),
            ),
            UnaryOperator::Reference => {
                todo!("reference operator lowering to llvm is not implemented")
            }
            UnaryOperator::Dereference => match kind {
                ValueKind::Indirect => {
                    expr.into_pointer_value();
                    (kind, expr)
                }
                ValueKind::Direct => (ValueKind::Indirect, expr),
            },
        };
        match indirect_value {
            Some(ptr) => {
                self.builder.build_store(ptr, value);
                None
            }
            None => Some(Value::new(kind, value)),
        }
    }

    fn fold_path_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        path: &Path,
        resolver: Resolver,
    ) -> Option<Value<'context>> {
        let item = resolver
            .resolve_path_in_value_namespace(self.db.upcast(), path)
            .unwrap();

        match item {
            ValueNamespaceItem::Function(_) => todo!(),
            ValueNamespaceItem::ValueConstructor(_) => todo!(),
```

```
            ValueNamespaceItem::LocalBinding(pattern_id) => match &self.body.patterns[pattern_id] {
                Pattern::Deconstructor(_, _) => todo!(),
                Pattern::Bind(name) => {
                    let value = self
                        .binding_stack
                        .borrow()
                        .get(name)
                        .expect("missing id {name:?} from scope.");
                    match indirect_value {
                        Some(ptr) => {
                            match value.kind {
                                    ValueKind::Indirect => {
                                        let source = value.value.into_pointer_value();
                                        let _ = self.builder.build_memcpy(
                                            ptr,
                                            4,
                                            source,
                                            4,
                                          self.context.i8_type().const_int(16, false),
                                        );
                                    }
                                    ValueKind::Direct => panic!("trying to put direct value into indirect value at bind {name:?}."),
                                }
                            None
                        }
                        None => Some(value.clone()),
                    }
                }
            },
        }
    }

    fn fold_literal_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        literal_id: TermId,
        literal: &Literal,
    ) -> Option<Value<'context>> {
        let literal_type = &self.inference.type_of_expression[literal_id];
        let value = match literal_type {
            Type::AbstractDataType(_) => todo!(),
            Type::FunctionDefinition(_) => todo!(),
            Type::Scalar(ScalarType::Integer(kind)) => {
                let value = match literal {
                    Literal::Integer(value, _) => *value,
                    _ => panic!(),
```

```
                };
                match kind {
                    IntegerKind::I32 => {
                        self.context.i32_type().const_int(value as u64, true).into()
                    }
                    IntegerKind::I64 => {
                        self.context.i64_type().const_int(value as u64, true).into()
                    }
                }
            }
            Type::Scalar(ScalarType::Boolean) => match literal {
                Literal::Bool(bool) => match bool {
                    true => self.context.bool_type().const_all_ones().into(),
                    false => self.context.bool_type().const_zero().into(),
                },
                _ => panic!(),
            },
            Type::Pointer(_) => todo!(),
        };
        match indirect_value {
            Some(ptr) => {
                self.builder.build_store(ptr, value);
                None
            }
            None => Some(Value::new(ValueKind::Direct, value)),
        }
    }

    fn fold_new_expression(
        &self,
        indirect_value: Option<PointerValue<'context>>,
        expr: TermId,
    ) -> Option<Value<'context>> {
        let ty = &self.inference.type_of_expression[expr];
        let ty = self.type_cache.llvm_type(ty);
        let ptr = self.builder.build_malloc(ty, "").unwrap();
        let _ = self.fold_expression(Some(ptr), expr).is_none();
        match indirect_value {
            Some(return_ptr) => {
                let i8_ptr_type =
                ↪   self.context.i8_type().ptr_type(AddressSpace::Generic);
                let source = self
                    .builder
                    .build_bitcast(ptr, i8_ptr_type, "")
                    .into_pointer_value();
              let source_type = &ptr.get_type().get_element_type().as_any_type_enum();
                let source_alignment =
                ↪   self.type_cache.target_data().get_abi_alignment(source_type);
```

```
                let sink = self
                    .builder
                    .build_bitcast(return_ptr, i8_ptr_type, "")
                    .into_pointer_value();
                let sink_alignment = self.type_cache.target_data().get_abi_alignment(
                    &return_ptr.get_type().get_element_type().as_any_type_enum(),
                );
                let size = self.type_cache.target_data().get_abi_size(source_type);
                let size = self
                    .context
                    .ptr_sized_int_type(self.type_cache.target_data(), None)
                    .const_int(size, false);
                let _ =
                    self.builder
                        .build_memcpy(sink, sink_alignment, source, source_alignment,
                        ↪  size);
                None
            }
            None => Some(Value::new(ValueKind::Direct, ptr.as_basic_value_enum())),
        }
    }
}
```

REFERENCES

AHO, Alfred V. et al. **Compilers: Principles, Techniques, and Tools (2Nd Edition)**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

APPEL, Andrew W.; PALSBERG, Jens. **Modern Compiler Implementation in Java**. 2nd. USA: Cambridge University Press, 2003. ISBN 052182060X.

CYTRON, Ron et al. Efficiently computing static single assignment form and the control dependence graph. **ACM Trans. Program. Lang. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 13, n. 4, p. 451–490, Oct. 1991. ISSN 0164-0925. DOI: 10.1145/115372.115320. Available from: <`https://doi.org/10.1145/115372.115320`>.

DENISOV, Alex; PANKEVICH, Stanislav. Mull It Over: Mutation Testing Based on LLVM. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). [S.l.]: IEEE, Apr. 2018. DOI: 10.1109/icstw.2018.00024. Available from: <`http://dx.doi.org/10.1109/ICSTW.2018.00024`>.

FORD, Bryan. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 37, n. 9, p. 36–47, Sept. 2002. ISSN 0362-1340. DOI: 10.1145/583852.581483. Available from: <`https://doi.org/10.1145/583852.581483`>.

________. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 1, p. 111–122, Jan. 2004. ISSN 0362-1340. DOI: 10.1145/982962.964011. Available from: <`https://doi.org/10.1145/982962.964011`>.

FROST, Richard A.; HAFIZ, Rahmatullah; CALLAGHAN, Paul. Parser Combinators for Ambiguous Left-Recursive Grammars. In________. **Practical Aspects of Declarative Languages**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. P. 167–181. ISBN 978-3-540-77442-6.

HIRSZ, Maciej. **Logos: Create ridiculously fast Lexers**. [S.l.: s.n.], 2020. accessed September 28, 2021. Available from: <`https://github.com/maciejhirsz/logos`>.

HOPCROFT, John E.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. **Introduction to Automata Theory, Languages and Computation**. 3. ed. Boston, MA: Pearson Addison-Wesley, 2007. ISBN 978-0-321-51448-6.

JHALA, Ranjit; VAZOU, Niki. **Refinement Types: A Tutorial**. [S.l.: s.n.], 2020. arXiv: 2010.07763 `[cs.PL]`.

KAZEROUNIAN, Milod et al. Refinement Types for Ruby. In: DILLIG, Isil; PALSBERG, Jens (Eds.). **Verification, Model Checking, and Abstract Interpretation**. Cham: Springer International Publishing, 2018. P. 269–290. ISBN 978-3-319-73721-8.

KLADOV, Aleksey. **Challenging LR Parsing**. [S.l.: s.n.], 2020. accessed September 28, 2021. Available from: <`https://rust-analyzer.github.io/blog/2020/09/16/challeging-LR-parsing.html`>.

________. **Simple but Powerful Pratt Parsing**. [S.l.: s.n.], 2020. accessed September 28, 2021. Available from: <`https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html`>.

LEE, Juneyoung et al. Reconciling high-level optimizations and low-level code in LLVM. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 2, OOPSLA, Oct. 2018. DOI: 10.1145/3276495. Available from: <`https://doi.org/10.1145/3276495`>.

PARR, Terence. **The Definitive ANTLR 4 Reference**. 2. ed. Raleigh, NC: Pragmatic Bookshelf, 2013. ISBN 978-1-93435-699-9. Available from: <`https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/`>.

PIERCE, Benjamin C. **Types and Programming Languages**. 1st. [S.l.]: The MIT Press, 2002. ISBN 0262162091.

PRATT, Vaughan R. Top down Operator Precedence. In: PROCEEDINGS of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Boston, Massachusetts: Association for Computing Machinery, 1973. (POPL '73), p. 41–51. ISBN 9781450373494. DOI: 10.1145/512927.512931. Available from: <`https://doi.org/10.1145/512927.512931`>.

SAMMLER, Michael et al. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In: PROCEEDINGS of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Virtual, Canada: Association for Computing Machinery, 2021. (PLDI 2021), p. 158–174. ISBN 9781450383912. DOI: 10.1145/3453483.3454036. Available from: <`https://doi.org/10.1145/3453483.3454036`>.

SIPSER, M. **Introduction to the Theory of Computation**. [S.l.]: Cengage Learning, 2012. ISBN 9781133187790.

VAZOU, Niki; SEIDEL, Eric L.; JHALA, Ranjit. LiquidHaskell: Experience with Refinement Types in the Real World. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 49, n. 12, p. 39–51, Sept. 2014. ISSN 0362-1340. DOI: 10.1145/2775050.2633366. Available from: <`https://doi.org/10.1145/2775050.2633366`>.

VEKRIS, Panagiotis; COSMAN, Benjamin; JHALA, Ranjit. Refinement Types for TypeScript. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 6, p. 310–325, June 2016. ISSN 0362-1340. DOI: 10.1145/2980983.2908110. Available from: <`https://doi.org/10.1145/2980983.2908110`>.

WARTH, Alessandro; DOUGLASS, James R.; MILLSTEIN, Todd. Packrat Parsers Can Support Left Recursion. In: PROCEEDINGS of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. San Francisco, California, USA: Association for Computing Machinery, 2008. (PEPM '08), p. 103–110. ISBN 9781595939777. DOI: 10.1145/1328408.1328424. Available from: <`https://doi.org/10.1145/1328408.1328424`>.

Ekitai - A Programming Language with Refinement Types and LLVM front end implementation

Bernardo Ferrari [1]

[1] Departamento de Informática e Estatística –
Universidade Federal de Santa Catarina (UFSC)

`bernardo.mendonca@grad.ufsc.br`

Abstract. *This article summarizes the thesis on the design and implementation of Ekitai, a programming language that integrates refinement types with an LLVM-IR front end. The primary goal of Ekitai is to leverage refinement types to enhance type safety and optimization during code generation. This work explores both theoretical and practical aspects of incorporating refinement types, demonstrating how they can express precise invariants and guide optimization and verification processes within the compilation pipeline.*

Resumo. *todo*

1. Introduction

In the realm of compiler design, ensuring type safety and optimizing code generation are paramount. Refinement types offer a robust mechanism to embed precise invariants within type systems, enabling more accurate and secure code compilation. Ekitai is a programming language that exemplifies this approach by integrating refinement types with LLVM Intermediate Representation (LLVM-IR), aiming to improve both the safety and performance of compiled code [Aho et al. 2006].

2. Motivation and Goals

The thesis begins by addressing the motivation behind Ekitai. Traditional type systems, while powerful, often fall short in expressing detailed invariants about data. Refinement types extend these capabilities by allowing types to be annotated with predicates, which specify more precise constraints [Jhala and Vazou 2020]. The main goals of Ekitai include:

1. Enhancing type safety by using refinement types.
2. Improving optimization during code generation.
3. Demonstrating the practical integration of refinement types with LLVM-IR.

3. Methodology

The development of Ekitai involved several stages:

1. **Lexical and Syntax Analysis**: The initial phase focused on defining the language's syntax and creating lexical analyzers and parsers to process source code into abstract syntax trees (ASTs) [Appel and Palsberg 2003].

2. **Semantic Analysis**: This stage ensured that the parsed code adhered to the language's type rules, incorporating refinement types to check for more precise invariants.
3. **Intermediate Code Generation**: The final stage translated the ASTs into LLVM-IR, leveraging the refinement types to guide optimization and verification processes.

4. Key Features of Ekitai

4.1. Refinement Types

Ekitai's type system is enriched with refinement types, allowing for more expressive and precise type definitions. These types enable the compiler to perform advanced checks and optimizations by embedding logical predicates within types [Jhala and Vazou 2020]. For instance, a typical integer type can be refined to ensure it is always positive:

```
type PosInt = {v:Int | v > 0}
```

This refinement ensures that any variable of type `PosInt` is guaranteed to be positive, enabling more robust and error-free code.

4.2. LLVM-IR Integration

By integrating with LLVM-IR, Ekitai benefits from LLVM's powerful optimization and code generation capabilities. This integration demonstrates how refinement types can be used to guide optimizations such as bounds checking elimination and nullability checks removal [Appel and Palsberg 2003].

4.3. Bidirectional Typing

Ekitai employs a bidirectional typing system, combining synthesis and checking judgments to ensure the correctness of programs. This system provides a robust framework for type inference and checking, enhancing both flexibility and safety [Pierce 2002]. The bidirectional system allows the compiler to infer types where possible and check them against defined constraints, ensuring code correctness.

5. Future Work

Future enhancements for Ekitai include:

1. **Predicate-Based Optimizations**: Extending the use of refinement predicates for more sophisticated optimizations in the AST and LLVM-IR.
2. **Extended Refinement Type System**: Supporting more complex constraints and type relationships to provide additional safety guarantees. This includes integrating more sophisticated SMT solvers to handle complex predicates.
3. **Improved Error Reporting**: Enhancing the compiler's error messages to be more informative and user-friendly, helping developers quickly understand and fix issues related to refinement types.
4. **Benchmarking and Real-World Testing**: Conducting extensive real-world testing to identify performance bottlenecks and guide further improvements. This involves testing Ekitai with large-scale projects to ensure its scalability and robustness.

6. Conclusion

Ekitai represents a significant step forward in leveraging refinement types for safer and more optimized code generation. By integrating these types with LLVM-IR, it showcases the potential for advanced type systems to influence modern compiler design positively. The insights gained from this work provide a strong foundation for future research and development in this area.

Refinement types enable more precise type checking, which can catch more errors at compile time and reduce the need for runtime checks. This leads to both safer and more efficient code. Additionally, the integration with LLVM-IR demonstrates how refinement types can be used in practical compiler implementations to achieve significant optimizations.

For more detailed information, the full thesis can be accessed through the provided references, offering a comprehensive overview of the design, implementation, and evaluation of Ekitai.

Referências

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools.* Pearson Education.

Appel, A. W. and Palsberg, J. (2003). *Modern Compiler Implementation in C/Java/ML.* Cambridge University Press.

Jhala, R. and Vazou, N. (2020). Refinement types. *Foundations and Trends® in Programming Languages*, 7(1-2):1–160.

Pierce, B. C. (2002). *Types and Programming Languages.* MIT Press.