

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.

Profª. 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

Profª. 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. Intermediate Code Representation. Compiler. LLVM. LLVM-IR.

LIST OF FIGURES

Figure 1 – Phases of a compiler and the intermediate representations between them. Adapted from Aho et al. (2006).	11
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.	21
Figure 3 – Visual representation of derivations (2.5) and (2.6) in its parse tree form.	23
Figure 4 – A context-free grammar after left factoring and removing left recursions from grammar (2.7).	25
Figure 5 – Predictive parsers for variables Expr (a), and Expr2 (b), from examples (2.8) and (2.9).	25
Figure 6 – Predictive parsers for variables Term (a), and Term2 (b), from examples (2.12) and (2.13).	25
Figure 7 – A predictive parser for variable Factor in (2.16).	26
Figure 8 – A Pratt parser for the ambiguous expression grammar (2.4).	27
Figure 9 – A sample concrete syntax tree for the grammar in Figure 4 representing sentence 5 + a .	29
Figure 10 – Data structures for a concrete parse tree and an equivalent abstract syntax tree.	30
Figure 11 – A tree transformation from the Expr and Expr2 nodes in Figure 10a into the AST nodes in Figure 10b.	30
Figure 12 – A tree transformation from the Term and Term2 nodes in Figure 10a into the Ast nodes in Figure 10b.	31
Figure 13 – A tree transformation from the NestedExpr , Number , and Id nodes in Figure 10a into the Ast nodes in Figure 10b.	31
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).	35
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).	37
Figure 16 – The inference rules for the three-place typing relation of the abstract syntax (2.28). Adapted from Pierce (2002).	39
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.	40

Figure 18 – An implementation of the <code>typeof</code> function for typechecking the simply typed lambda calculus described in Section 2.4.3 written in Rust.	41
Figure 19 – An intermediate representation for code generation. Adapted from Appel and Palsberg (2003).	43
Figure 20 – Abstract syntax for refinement predicates and constraints. Adapted from (JHALA; VAZOU, 2020).	46
Figure 21 – Abstract syntax for terms, types, and context of a simply typed lambda calculus with refinements. Adapted from (JHALA; VAZOU, 2020).	48
Figure 22 – Inference rules for the Well-formedness judgment. Adapted from Jhala and Vazou (2020).	49
Figure 23 – Inference rules for the entailment judgment. Adapted from Jhala and Vazou (2020).	49
Figure 24 – Inference rules for the Subtyping judgment. Adapted from Jhala and Vazou (2020).	50
Figure 25 – Inference rules for the Synthesis judgment. Adapted from Jhala and Vazou (2020).	51
Figure 26 – Inference rules for the Checking judgment. Adapted from Jhala and Vazou (2020).	51
Figure 27 – Subtyping function for the subtyping relation. Adapted from Jhala and Vazou (2020).	55
Figure 28 – Synthesis function for the synthesis relation. Adapted from Jhala and Vazou (2020).	56
Figure 29 – Synthesis function for the synthesis relation. Adapted from Jhala and Vazou (2020).	57
Figure 30 – Building block regular definitions for token patterns of Figure 31.	60
Figure 31 – Token names and patterns for the Ekitai lexer.	60
Figure 32 – Top level grammar productions for the Ekitai’s parser grammar.	61
Figure 33 – Expression grammar productions for the Ekitai’s parser grammar.	63
Figure 34 – Type grammar productions for the Ekitai’s parser grammar.	63
Figure 35 – Ekitai’s AST derived from the lambda calculus with refinements from Section 3.1.	64

CONTENTS

1	INTRODUCTION	10
1.1	MOTIVATION AND RESEARCH PROBLEM	14
1.2	GOALS	14
1.2.1	Specific Goals	14
1.3	METHODOLOGY	15
1.4	STRUCTURE OF THE WORK	15
2	A REVIEW ON FRONT END DESIGN AND IMPLEMENTATION	16
2.1	STRINGS AND LANGUAGES	16
2.2	LEXICAL ANALYSIS	17
2.3	SYNTAX ANALYSIS	20
2.3.1	Precise definition of context-free grammars and ambiguity	20
2.3.2	Review of top-down and bottom-up parsers	24
2.3.3	Semantic actions and syntax-directed translations	28
2.3.4	Abstract syntax trees	29
2.3.5	Syntax error recovery	31
2.4	SEMANTIC ANALYSIS	32
2.4.1	Operational Semantics	34
2.4.2	The typing relation	36
2.4.3	Pure simply typed lambda calculus	38
2.4.4	A type checker implementation for the simply typed lambda calculus	40
2.5	INTERMEDIATE CODE GENERATION	42
2.5.1	Three-address code	42
2.5.2	Single Static Assignment (SSA)	43
2.5.2.1	<i>Key Characteristics of SSA</i>	44
3	REFINEMENT TYPES WITH PREDICATES	45
3.1	EXTENDING THE SIMPLY TYPED LAMBDA CALCULUS WITH REFINEMENTS	46
3.1.1	Abstract syntax of predicates and constraints	46
3.1.2	Abstract syntax of terms and types	47
3.1.3	Variable substitution rules for refinement types	48
3.1.4	Judgments	48
3.1.5	Bidirectional typing	50
3.1.6	An implementation of a verification condition generator	54

4	THE LLVM INTERMEDIATE REPRESENTATION	58
4.1	LLVM'S INTERMEDIATE REPRESENTATION (IR)	58
4.1.1	Characteristics of LLVM IR	58
4.1.2	Optimizations in LLVM	58
5	THE EKITAI LANGUAGE	60
5.1	THE EKITAI'S LEXER	60
5.2	THE EKITAI'S PARSER	61
5.3	THE EKITAI'S TYPE SYSTEM	63
5.3.1	Abstract Syntax Tree (AST)	64
5.3.2	Predicates and Constraints	66
5.3.3	Type Inference and Checking	66
5.3.4	Entailment and Substitution	68
5.3.5	Interfacing with the SMT Solver	68
5.4	THE EKITAI'S LLVM INTERMEDIATE CODE GENERATOR . .	69
5.5	EXAMPLES AND INTERMEDIATE CONSTRUCTS	74
5.5.1	Identity function with refinements	74
5.5.2	Absolute value function with refinements	76
6	CONCLUSION	96
6.1	FINAL CONSIDERATIONS	96
6.2	FUTURE WORK	96
A	EKITAI'S IMPLEMENTATION	98
A.1	SUBMODULE HIR::LIQUID	98
A.2	MODULE CODEGEN	121
	References	162

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-level 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.

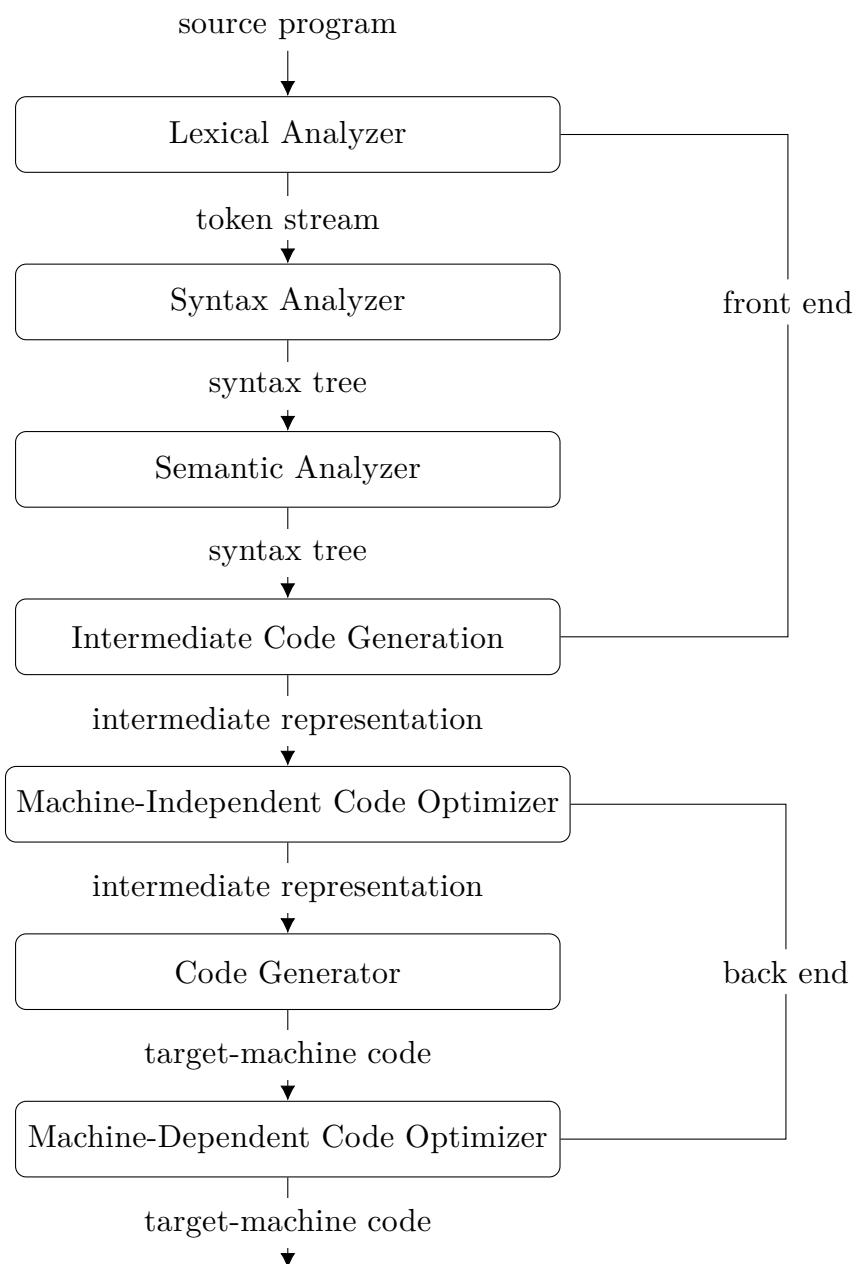


Figure 1 – Phases of a compiler and the intermediate representations between them. Adapted from Aho et al. (2006).

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

$$x * 30 \quad (1.1)$$

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

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

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

$$x : \text{Bool} \quad \text{and} \quad 30 : \text{Int}$$

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 seem 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

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

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

$$\begin{aligned} u, v, s, w &\in \Sigma^* \\ u &= u_1 u_2 u_3 \dots u_i \quad \text{where } u_1 u_2 u_3 \dots u_i \in \Sigma \quad \text{and } i \geq 0 \\ v &= v_1 v_2 v_3 \dots v_j \quad \text{where } v_1 v_2 v_3 \dots v_j \in \Sigma \quad \text{and } j \geq 0 \\ s &= u_1 u_2 u_3 \dots u_i v_1 v_2 v_3 \dots v_j \\ u\epsilon &= u \quad \epsilon u = u \\ (uv)w &= u(vw) \end{aligned}$$

Next, we perform the inductive construction

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

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:

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

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., `{}`).

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:

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

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 $L(\epsilon) = \{\epsilon\}$, 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 $s_1 \mid s_2 \mid \dots \mid s_k$ can be abbreviated as $[s_1 s_2 \dots s_k]$ (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:

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

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 $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ 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

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

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

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

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 not 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

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

and the top one, denoting the whole expression

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

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

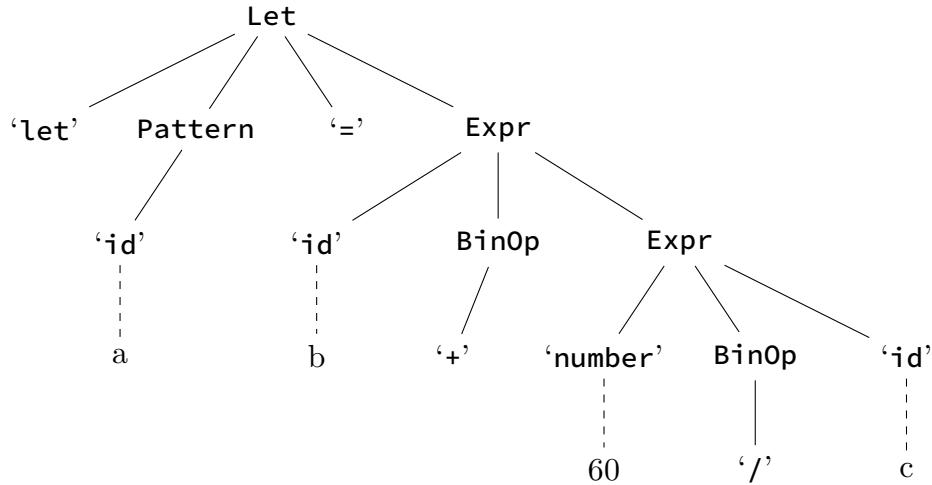


Figure 2 – A possible parse tree output for the token stream (2.1). Leafs are shown as the token names from the token stream. Here lexemes for non-trivial tokens are appended with dashed lines underneath token names for example purposes.

- 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

Variable ::= 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

$$\begin{aligned}
 \text{Expr} &::= \text{Expr} \text{ BinOp } \text{Expr} \\
 &\quad | \text{ '(' Expr ')'} \\
 &\quad | \text{ 'number'} \\
 &\quad | \text{ 'id'} \\
 \text{BinOp} &::= '+' | '-' | '*' | '/'
 \end{aligned} \tag{2.4}$$

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

$$V ::= s_1$$

$$V ::= s_2$$

into

$$V ::= s_1 | s_2$$

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 $s_1, s_2, s_3, \dots, s_k \in (\Sigma \cup V)^*$ for $k \geq 0$ where

$$u \Rightarrow s_1 \Rightarrow s_2 \Rightarrow s_3 \Rightarrow \dots \Rightarrow s_k \Rightarrow v$$

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 $s = \mathbf{id} + \mathbf{number} * \mathbf{id}$ 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

Expr

then choose a production with **Expr** as head and substitute **Expr** 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

Expr \Rightarrow **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

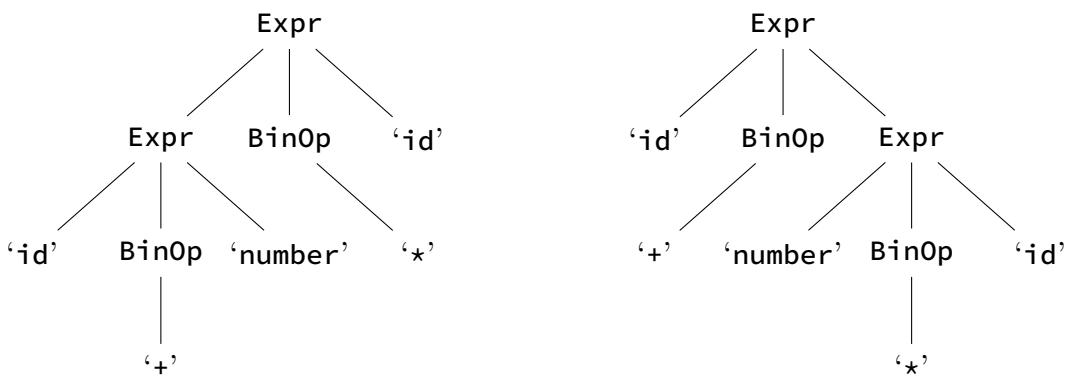
$$\begin{aligned}
 \mathbf{Expr} &\Rightarrow \mathbf{Expr} \mathbf{BinOp} \mathbf{Expr} \\
 &\Rightarrow \mathbf{Expr} \mathbf{BinOp} \mathbf{Expr} \mathbf{BinOp} \mathbf{Expr} \\
 &\Rightarrow 'id' \mathbf{BinOp} \mathbf{Expr} \mathbf{BinOp} \mathbf{Expr} \\
 &\Rightarrow 'id' '+' \mathbf{Expr} \mathbf{BinOp}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

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

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

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

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

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 tree 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 shown 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

$$\begin{aligned}
 \text{Expr} &::= \text{Term Expr2} & (2.8) \\
 \text{Expr2} &::= '+' \text{ Term Expr2} & (2.9) \\
 &\quad | '-' \text{ Term Expr2} & (2.10) \\
 &\quad | \varepsilon & (2.11) \\
 \text{Term} &::= \text{Factor Term2} & (2.12) \\
 \text{Term2} &::= '*' \text{ Factor Term2} & (2.13) \\
 &\quad | '/' \text{ Factor Term2} & (2.14) \\
 &\quad | \varepsilon & (2.15) \\
 \text{Factor} &::= '(' \text{ Expr } ')' & (2.16) \\
 &\quad | \text{'number'} & (2.17) \\
 &\quad | \text{'id'} & (2.18)
 \end{aligned}$$

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.

```

1  fn parse_Expr2(p: Parser) -> Parse {
2    if p.at("+" | "-") {
3      let op = p.eat_token();
4      let t = parse_Term(p);
5      let e2 = parse_Expr2(p);
6      Expr2(op, t, e2)
7    } else {
8      Empty
9    }
10 }

1 fn parse_Expr(p: Parser) -> Parse {
2   let t = parse_Term(p);
3   let e2 = parse_Expr2(p);
4   Expr(t, e2)
5 }
```

(a)

(b)

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

```

1  fn parse_Term2(p: Parser) -> Parse {
2    if p.at("*" | "/") {
3      let op = p.eat_token();
4      let f = parse_Factor(p);
5      let t2 = parse_Term2(p);
6      Term2(op, f, t2)
7    } else {
8      Empty
9    }
10 }

1 fn parse_Term(p: Parser) -> Parse {
2   let f = parse_Factor(p);
3   let t2 = parse_Term2(p);
4   Term(f, t2)
5 }
```

(a)

(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

```

1  fn parse_Factor(p: Parser) -> Parse {
2    if p.at("(") {
3      p.eat_token();
4      let expr = parse_Expr(p);
5      p.expect(")");
6      NestedExpr(expr)
7    } else if p.at("id") {
8      let token = p.eat_token();
9      Id(token)
10 } else if p.at("number") {
11   let token = p.eat_token();
12   Number(token)
13 } else {
14   panic!("Parse error.");
15 }
16 }
```

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 `id1 + number * id2` it:

- parses `id1` 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 `id2` 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 `id2` and name it ‘right’; then, we construct a node `Expr` with ‘left’, which is `number` for this iteration, the operator `*`, and ‘right’, which is `id2`;
- we then return again with another `Expr` node with ‘left’, which now is `id1`, the operator `+`, and the last returned expression as ‘right’ finally forming the parse tree with correct precedence found at Figure 3b.

```

1  fn parse_Expr(p: Parser) -> Parse {
2      parse_Expr_binding_power(p, 0)
3  }
4  fn parse_Expr_binding_power(p: Parser, min_bp: Int) -> Parse {
5      let left = if p.at("id") {
6          let token = p.eat_token();
7          Id(token)
8      } else if p.at("number") {
9          let token = p.eat_token();
10         Number(token)
11     } else if p.at("(") {
12         p.eat_token();
13         let expr = p.parse_Expr();
14         p.expect(")");
15         expr
16     } else {
17         panic!("Parse error.");
18     };
19     loop {
20         let op = if p.at("+") { Sum }
21             else if p.at("-") { Minus }
22             else if p.at("*") { Times }
23             else if p.at("/") { Div }
24             else { break; }
25         let (left_bp, right_bp) = binding_power(op);
26         if left_bp < min_bp {
27             break;
28         }
29         p.eat_token();
30         let right = parse_Expr_binding_power(p, right_bp);
31         left = Expr(left, op, right);
32     }
33     left
34 }
35 fn binding_power(op: BinOp) -> (Int, Int) {
36     match op {
37         Sum | Minus => (1, 2),
38         Times | Div => (3, 4),
39     }
40 }
```

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 guarantees 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 hand-written 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

$$\text{Expr} ::= \text{Term Expr2} \{ \text{Expr.n} = \text{Expr}(\text{Term.n}, \text{Expr2.n}) \} \quad (2.19)$$

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 authors 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.

```

1   Expr
2     Term
3       Factor
4         Number "5"
5   Expr2
6     Plus "+"
7     Term
8       Factor
9         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.

```

1  enum Parse {
2      Expr(Parse, Parse),
3      Expr2(Token, Parse, Parse),
4      Term(Parse, Parse),
5      Term2(Token, Parse, Parse),
6      Number(Token),
7      Id(Token),
8      Empty,
9  }
10 }
```

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

```

1  enum Ast {
2      Expr(Ast, BinOp, Ast),
3      Number(Int),
4      Id(String),
5  }
6  enum BinOp {
7      Plus,
8      Minus,
9      Times,
10     Div,
11 }
```

(b) Abstract syntax tree definition.

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

```

1  fn translate_expr(expr: Parse) -> Ast {
2      match expr {
3          Parse::Expr(term, cont) => {
4              let ast_term = translate_term(term);
5              match translate_expr2(cont) {
6                  None => ast_term,
7                  Some((op, ast_cont)) => Ast::Expr(ast_term, op, ast_cont),
8              }
9          }
10         _ => panic!("not an Expr"),
11     }
12 }
13 fn translate_expr2(expr2: Parse) -> Option<(BinOp, Ast)> {
14     match expr2 {
15         Parse::Expr2(op_tok, term, cont) => {
16             let op = into_binop(op_tok);
17             let ast_term = translate_term(term);
18             let ast = match translate_expr2(cont) {
19                 None => (op, ast_term),
20                 Some(op2, ast_term2) => (op, Ast::Expr(ast_term, op2, ast_term2)),
21             };
22             Some(ast)
23         }
24         Parse::Empty => None,
25         _ => panic!("not an Expr2"),
26     }
27 }
```

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

```

1  fn translate_term(term: Parse) -> Ast {
2      match term {
3          Parse::Term(factor, cont) => {
4              let ast_factor = translate_factor(factor);
5              match translate_factor2(cont) {
6                  None => ast_factor,
7                  Some((op, ast_cont)) => Ast::Expr(ast, op, ast_cont),
8              }
9          }
10         _ => panic!("not an Term"),
11     }
12 }
13 fn translate_term2(term2: Parse) -> Option<(BinOp, Ast)> {
14     match term2 {
15         Parse::Term2(op_tok, factor, cont) => {
16             let op = into_binop(op_tok);
17             let ast_factor = translate_factor(factor);
18             let ast = match translate_term2(cont) {
19                 None => (op, ast_factor),
20                 Some(op2, ast_factor2) => {
21                     (op, Ast::Expr(ast_factor, op2, ast_factor2))
22                 }
23             };
24             Some(ast)
25         }
26         Parse::Empty => None,
27         _ => panic!("not an Term2"),
28     }
29 }
```

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

```

1  fn translate_factor(factor: Parse) -> Ast {
2      match factor {
3          Parse::NestedExpr(expr) => translate_expr(expr),
4          Parse::Number(token) => Ast::Number(token.into_int()),
5          Parse::Id(token) => Ast::Id(token.into_string()),
6          _ => panic!("not an Factor"),
7      }
8  }
```

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

hoping 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 start reasoning about the semantic aspects of a language suppose the following abstract syntax

```
term ::= true
      | false
      | if term then term else term
      | 0
      | succ term
      | pred term
      | iszero term (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

$$\{\text{true}, \text{false}, \theta\} \subseteq T \quad (2.21)$$

$$\text{if } \text{term} \in T, \text{ then } \{\text{succ term}, \text{pred term}, \text{iszzero term}\} \subseteq T \quad (2.22)$$

$$\text{if } \text{term}_1, \text{term}_2, \text{term}_3 \in T, \text{ then if } \text{term}_1 \text{ then term}_2 \text{ else term}_3 \in T \quad (2.23)$$

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:

$$\begin{array}{c} \text{true} \in T \qquad \qquad \text{false} \in T \qquad \qquad \theta \in T \\ \\ \frac{\text{term} \in T}{\text{succ term} \in T} \qquad \frac{\text{term} \in T}{\text{pred term} \in T} \qquad \frac{\text{term} \in T}{\text{pred term} \in T} \\ \\ \frac{\text{term}_1 \in T \qquad \text{term}_2 \in T \qquad \text{term}_3 \in T}{\text{if term}_1 \text{ then term}_2 \text{ else term}_3 \in T} \end{array}$$

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

```
if true than succ 0 else pred succ 0
```

is a **term** as follows:

$$\begin{array}{c}
 \frac{\text{true} \in T}{\text{if true than succ } 0 \text{ else pred succ } 0 \in T} \\
 \frac{\theta \in T}{\text{succ } \theta \in T} \\
 \frac{\theta \in T}{\text{pred succ } \theta \in T} \\
 \frac{\theta \in T}{\text{succ } \theta \in T} \\
 \frac{\theta \in T}{\text{pred succ } \theta \in T}
 \end{array}$$

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

$$\begin{aligned}
 \text{value} &::= \text{true} \\
 &\quad | \quad \text{false} \\
 &\quad | \quad \text{nvalue} \\
 \text{nvalue} &::= 0 \\
 &\quad | \quad \text{succ nvalue}
 \end{aligned} \tag{2.24}$$

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

$$\text{term} \rightarrow \text{term}_1$$

meaning **term** evaluates to **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

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

$$\begin{array}{c}
 \frac{\text{if true then term}_1 \text{ else term}_2 \rightarrow \text{term}_1}{\text{if false then term}_1 \text{ else term}_2 \rightarrow \text{term}_2} \text{Eval-IfTrue} \\
 \\
 \frac{\text{term}_1 \rightarrow \text{term}_4}{\text{if term}_1 \text{ then term}_2 \text{ else term}_3 \rightarrow \text{if term}_4 \text{ then term}_2 \text{ else term}_3} \text{Eval-If} \\
 \\
 \frac{\text{term}_1 \rightarrow \text{term}_2}{\text{succ term}_1 \rightarrow \text{succ term}_2} \text{Eval-Succ} \\
 \\
 \frac{}{\text{pred succ nvalue} \rightarrow \text{nvalue}} \text{Eval-PredSucc} \\
 \\
 \frac{\text{term}_1 \rightarrow \text{term}_2}{\text{pred term}_1 \rightarrow \text{pred term}_2} \text{Eval-Pred} \\
 \\
 \frac{}{\text{iszzero } 0 \rightarrow \text{true}} \text{Eval-IszzeroZero} \\
 \\
 \frac{\text{term}_1 \rightarrow \text{term}_2}{\text{iszzero term}_1 \rightarrow \text{iszzero term}_2} \text{Eval-Iszzero}
 \end{array}$$

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

$$\frac{\text{iszzero } 0 \rightarrow \text{true}}{\text{if iszzero } 0 \text{ then succ } 0 \text{ else false} \rightarrow \text{if true then succ } 0 \text{ else false}} \text{Eval-If}$$

proving that

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

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

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

thus proving that

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

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

$$\frac{\overline{\quad} ?}{\overline{0 \rightarrow ?}} \text{Eval-Succ}$$

$$\overline{\text{succ } 0 \rightarrow ?}$$

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

```
if iszero 0 then succ 0 else false → 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

$$\begin{array}{c} \text{Type} ::= \text{Nat} \\ | \text{Bool} \end{array} \quad (2.27)$$

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

$$\begin{array}{c} \frac{}{\text{true} : \text{Bool}} \text{Ty-True} \quad \frac{}{\text{false} : \text{Bool}} \text{Ty-False} \\ \\ \frac{\text{term}_1 : \text{Bool} \quad \text{term}_2 : \text{Type} \quad \text{term}_3 : \text{Type}}{\text{if term}_1 \text{ then term}_2 \text{ else term}_3 : \text{Type}} \text{Ty-If} \\ \\ \frac{\text{t}_1 : \text{Nat}}{\text{succ t}_1 : \text{Nat}} \text{Ty-Succ} \quad \frac{\text{t}_1 : \text{Nat}}{\text{pred t}_1 : \text{Nat}} \text{Ty-Pred} \quad \frac{\text{t}_1 : \text{Nat}}{\text{iszzero t}_1 : \text{Bool}} \text{Ty-Iszero} \end{array}$$

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

iszzero succ 0 : Bool

by applying the following sequence of inferences

$$\begin{array}{c} \frac{}{\text{0} : \text{Nat}} \text{Ty-Zero} \\ \frac{\text{0} : \text{Nat}}{\text{succ 0 : Nat}} \text{Ty-Succ} \\ \frac{\text{0} : \text{Nat}}{\text{iszzero succ 0 : Bool}} \text{Ty-IsZero} \end{array}$$

but, if we try to prove something wrong like

succ true : Num or 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

$$\frac{\frac{\frac{}{\text{0} : \text{Nat}} \text{Ty-Zero}}{\frac{\text{0} : \text{Nat}}{\text{succ 0 : Nat}}} \text{Ty-Succ} \quad \frac{}{\text{false : Bool}}}{\frac{\text{0} : \text{Nat} \quad \text{succ 0 : Nat} \quad \text{false : Bool}}{\text{if iszzero 0 the succ 0 else false : ?}}} \text{Ty-If}$$

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

```

term ::= true
      | false
      | if term then term else term
      | 0
      | succ term
      | pred term
      | iszzero term
      | var
      | let var = term in term
      |  $\lambda$ var : Type. term
      | term term

var ::= a | b | c | ... | x | y | z | aa | ab | ...

Type ::= Bool
      | Nat
      | Type → Type

```

(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

```

 $\Gamma$  ::=  $\Gamma, \text{var} : \text{Type}$ 
      |  $\emptyset$ 

```

(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 $\Gamma \vdash \text{term} : \text{Type}$, and it means that **term** has type **Type** under the context Γ (PIERCE, 2002). If the context is empty we write $\emptyset \vdash \text{term} : \text{Type}$, but the \emptyset is usually omitted as in $\vdash \text{term} : \text{Type}$.

Pierce (2002) defines the three-place typing relation inference rules as shown in Figure 16. Supposing all typing relation rules from Figure 15 were updated to the

$$\begin{array}{c}
 \frac{\text{var} : \text{Type} \in \Gamma}{\Gamma \vdash \text{var} : \text{Type}} \text{Ty-Var} \\
 \\
 \frac{\Gamma \vdash \text{term}_1 : \text{Type}_1 \quad \Gamma, \text{var} : \text{Type}_1 \vdash \text{term}_2 : \text{Type}_2}{\Gamma \vdash \text{let var} = \text{term}_1 \text{ in term}_2 : \text{Type}_2} \text{Ty-Let} \\
 \\
 \frac{\Gamma, \text{var} : \text{Type}_1 \vdash \text{term} : \text{Type}_2}{\Gamma \vdash \lambda \text{var} : \text{Type}_1. \text{term} : \text{Type}_1 \rightarrow \text{Type}_2} \text{Ty-FnAbs} \\
 \\
 \frac{\Gamma \vdash \text{term}_1 : \text{Type}_1 \rightarrow \text{Type}_2 \quad \Gamma \vdash \text{term}_2 : \text{Type}_1}{\Gamma \vdash \text{term}_1 \text{ term}_2 : \text{Type}_2} \text{Ty-FnApp}
 \end{array}$$

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

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

```
let fun = λx : Nat.iszero x in fun succ 0
```

as presented in the following inference trees

$$\begin{array}{c}
 \frac{x : \text{Nat} \in \Gamma}{\Gamma, x : \text{Nat} \vdash x : \text{Nat}} \text{Ty-Var} \\
 \\
 \frac{\Gamma, x : \text{Nat} \vdash \text{iszero } x : \text{Bool}}{\Gamma, x : \text{Nat} \vdash \lambda x : \text{Nat}. \text{iszero } x : \text{Nat} \rightarrow \text{Bool}} \text{Ty-Iszero} \\
 \\
 \frac{\text{fun} : \text{Nat} \rightarrow \text{Nat} \in \Gamma}{\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Nat} \vdash \text{fun} : \text{Nat} \rightarrow \text{Nat}} \text{Ty-Var} \\
 \\
 \frac{\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Nat} \vdash 0 : \text{Nat} \in \Gamma}{\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Nat} \vdash \lambda x : \text{Nat}. \text{succ } x : \text{Nat} \rightarrow \text{Nat}} \text{Ty-Zero} \\
 \\
 \frac{\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Nat} \vdash \lambda x : \text{Nat}. \text{succ } x : \text{Nat} \rightarrow \text{Nat}}{\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Nat} \vdash \text{succ } 0 : \text{Nat}} \text{Ty-Succ}
 \end{array}$$

$\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Bool} \vdash \text{fun} : \text{Nat} \rightarrow \text{Bool}$	
$\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Bool} \vdash \text{succ } 0 : \text{Nat}$	Ty-FnApp
$\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Bool} \vdash \text{fun succ } 0 : \text{Bool}$	
$\Gamma \vdash \lambda x : \text{Nat}. \text{iszzero } x : \text{Nat} \rightarrow \text{Bool}$	
$\Gamma, \text{fun} : \text{Nat} \rightarrow \text{Bool} \vdash \text{fun succ } 0 : \text{Bool}$	Ty-Let
$\Gamma \vdash \text{let fun} = \lambda x : \text{Nat}. \text{iszzero } x \text{ in fun succ } 0 : \text{Bool}$	

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

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

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

`typeof(Γ , term) = 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 shown by Figure 17.

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 shown 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 considered to be well-typed and sound to the language's type system.

```

1  fn typeof(cx: Context, t: Term) -> Type {
2    match t {
3      True => Bool,
4      False => Bool,
5      If(t1, t2, t3) => if typeof(cx, t1) == Bool {
6        let if_ty = typeof(cx, t2);
7        if if_ty == typeof(cx, t3) {
8          if_ty
9        } else {
10          panic!("if branches differ in type")
11        }
12      } else {
13        panic!("if condition term not a Bool")
14      },
15      Zero => Type::Num,
16      Succ(t) => if typeof(cx, t) == Type::Num {
17        Type::Num
18      } else {
19        panic!("succ argument term not a Num")
20      },
21      Pred(t) => if typeof(cx, t) == Type::Num {
22        Type::Num
23      } else {
24        panic!("pred argument term not a Num")
25      },
26      IsZero(t) => if typeof(cx, t) == Type::Num {
27        Type::Bool
28      } else {
29        panic!("iszero argument term not a Num")
30      }
31      Var(x) => get_type_in_context(cx, x)
32      Let(x, t1, t2) => {
33        let x_ty = typeof(t1);
34        let cx1 = add_binding(cx, (x, x_ty));
35        typeof(cx1, t2)
36      }
37      Fn(x, x_ty, t) => {
38        let cx1 = add_binding(cx, (x, x_ty));
39        Type::Fn(ty, typeof(cx1, t))
40      }
41      App(t1, t2) => match typeof(cx, t1) {
42        Fn(in_ty, out_ty) => if typeof(cx, t2) == in_ty {
43          out_ty
44        } else {
45          panic!("Application argument term type differs from
46            ↳ function's input type")
47        },
48        _ => panic!("Application function term not a Function type")
49      }
50    }

```

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 single 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 x_1 , x_2 , 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 $x_3 = \phi(x_1, x_2)$.
- **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 where 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

$p ::= x \mid y \mid z \mid \dots$	<i>Variables</i>
true false	<i>Booleans</i>
0 -1 1 ...	<i>Numbers</i>
$p_1 + p_2 \mid p_1 - p_2 \mid p_1 * p_2 \mid \dots$	<i>Arithmetic Operations</i>
$p_1 \&& p_2$	<i>Conjunction</i>
$p_1 \mid\mid p_2$	<i>Disjunction</i>
$!p_1$	<i>Negation</i>
if p_1 then p_2 else p_3	<i>Conditional</i>
$p_1 \bowtie p_2$	<i>Interpreted Operations</i>
$f(p)$	<i>Uninterpreted Function</i>
$c ::= p$	<i>Predicate</i>
$c_1 \wedge c_2$	<i>Conjunction</i>
$\forall x : b. p \implies c$	<i>Implication</i>

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

used by the authors is a taken from the *quantifier-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

if p_1 **then** p_2 **else** p_3

was added as an abbreviation to

$$(p_1 \implies x = p_2) \wedge (\neg p_1 \implies x = p_3)$$

and the operation $p_1 \bowtie p_2$ 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 $\mathbf{0} \leq v$ to denote the natural numbers and $\mathbf{0} \leq v \ \&\& \ v = \text{length}(x)$ 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. Constraints are either a single quantifier-free predicate p , a conjunction $c_1 \wedge c_2$, 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 $c_i = \forall x.p_i \implies q_i$ such that $c \iff c_i$ (JHALA; VAZOU, 2020). The validity of each c_i is then determined by checking if the quantifier free predicate $p_i \wedge \neg q_i$ 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 $x:t_1 \rightarrow t_2$ composed of a binder x of type t_1 which can appear at the refinement predicate of t_2 .

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

$e ::= c$	<i>Constants</i>
x	<i>Variables</i>
$\text{if } x \text{ then } e \text{ else } e$	<i>If Expression</i>
$\text{let } x = e \text{ in } e$	<i>Let Binding</i>
$\text{rec } x = e:t \text{ in } e$	<i>Recursive Binding</i>
$\lambda x.e$	<i>Function</i>
$e x$	<i>Application</i>
$e:t$	<i>Type Annotation</i>
$t ::= b\{x p\}$	<i>Refined Base</i>
$x:t \rightarrow t$	<i>Dependent Function</i>
$b ::= \text{int}$	<i>Base Integer Type</i>
bool	<i>Base Boolean Type</i>
$\Gamma ::= \emptyset$	<i>Empty Context</i>
$\Gamma, x : t$	<i>Variable Binding</i>

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

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

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

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

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

$$\begin{aligned} (x:t_1 \rightarrow t_2)[x := y] &= x:t_1[x := y] \rightarrow t_2 \\ (z:t_1 \rightarrow t_2)[x := y] &= z:t_1[x := y] \rightarrow t_2[x := y] \end{aligned}$$

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 $\Gamma \vdash p$ meaning that in the context Γ the predicate p is *well-sorted*, that is, 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 $\Gamma \vdash t$ meaning that in the context Γ the type 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).

$$\frac{\Gamma, x : b \vdash p}{\Gamma \vdash b\{x|p\}} \text{ Wf-Base} \quad \frac{\Gamma \vdash t_1 \quad \Gamma, x : t_1 \vdash t_2}{\Gamma \vdash x : t_1 \rightarrow t_2} \text{ Wf-Fun}$$

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 $b\{x|p\}$ to be well-formed in a context Γ , the predicate p needs to be well-sorted in the context Γ extended with the binding $x : b$ (assuming the erasure of all refinements from Γ) (JHALA; VAZOU, 2020). And, the rule Wf-Fun defines that, for a function type $x : t_1 \rightarrow t_2$ to be well-formed, the type t_1 needs to be well-formed, and, the type t_2 needs to be well-formed with the context extended with the parameter $x : t_1$ (JHALA; VAZOU, 2020).

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

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

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 c given that 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 $\Gamma \vdash t_1 \prec t_2$ meaning that in the context Γ the type t_1 is a subtype of t_2 , that is, all values of t_1 are present in t_2 but the contrary may not be true (JHALA; VAZOU, 2020).

$$\begin{array}{c}
 \frac{\Gamma \vdash \forall x_1 : b. p_1 \implies p_2[x_2 := x_1]}{\Gamma \vdash b\{x_1|p_1\} \prec: b\{x_2|p_2\}} \text{ Sub-Base} \\
 \\
 \frac{\Gamma \vdash t_{2.1} \prec: t_{1.1} \quad \Gamma, x_2 : t_{2.1} \vdash t_{1.2} \prec: t_{2.2}}{\Gamma \vdash x_1 : t_{1.1} \rightarrow t_{1.2} \prec: x_2 : t_{2.1} \rightarrow t_{2.2}} \text{ Sub-Fun}
 \end{array}$$

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 $b\{x_1|p_1\}$ to be a subtype of $b\{x_2|p_2\}$, the context has to entail the constraint $\forall x_1 : b. p_1 \implies p_2[x_2 := x_1]$ which means that if p_1 holds then so must p_2 with all free occurrences of x_2 substituted by 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 $\Gamma \vdash e \Rightarrow t$ meaning that in the context Γ the type t can be generated, or synthesized, for the term 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 $\Gamma \vdash e \Leftarrow t$ meaning that in the context Γ the type t is a valid type for the term e , that is, constraining the term e to *synthesize* a type s where $s \prec: t$ (JHALA; VAZOU, 2020). This judgment is used to verify that a term is of given annotated type and to push the type annotation inside its 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 $\Gamma \vdash x : T$ 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.

$$\begin{array}{c} \frac{x : t \in \Gamma \quad \text{self}(x, t) = s}{\Gamma \vdash x \Rightarrow s} \text{Syn-Var} \quad \frac{\text{prim}(c) = t}{\Gamma \vdash c \Rightarrow t} \text{Syn-Const} \\ \\ \frac{\Gamma \vdash e \Rightarrow y:t_1 \rightarrow t_2 \quad \Gamma \vdash x \Leftarrow t_1}{\Gamma \vdash e \ x \Rightarrow t_2[y := x]} \text{Syn-App} \quad \frac{\Gamma \vdash t \quad \Gamma \vdash e \Leftarrow t}{\Gamma \vdash e:t \Rightarrow t} \text{Syn-Ann} \end{array}$$

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

$$\begin{array}{c} \frac{\Gamma \vdash e \Rightarrow t_1 \quad \Gamma \vdash t_1 \prec t_2}{\Gamma \vdash e \Leftarrow t_2} \text{Chk-Syn} \quad \frac{\Gamma, x : t_1 \vdash e \Leftarrow t_2}{\Gamma \vdash \lambda x. e \Leftarrow x:t_1 \rightarrow t_2} \text{Chk-Fun} \\ \\ \frac{\Gamma \vdash e_1 \Rightarrow t_1 \quad \Gamma, x : t_1 \vdash e_2 \Leftarrow t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Leftarrow t_2} \text{Chk-Let} \\ \\ \frac{\Gamma \vdash t_1 \quad \Gamma, x : t_1 \vdash e_1 \Leftarrow t_1 \quad \Gamma, x : t_1 \vdash e_2 \Leftarrow t_2}{\Gamma \vdash \text{rec } x = e_1:t_1 \text{ in } e_2 \Leftarrow t_2} \text{Chk-Rec} \\ \\ \frac{\Gamma, y : \text{int}\{y|x\} \vdash e_1 \Leftarrow t \quad \Gamma, y : \text{int}\{y|!x\} \vdash e_2 \Leftarrow t}{\Gamma \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 \Leftarrow t} \text{Chk-If} \end{array}$$

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 $\text{prim}(c)$ 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

```
prim(0) = int{x|x == 0}
prim(1) = int{x|x == 1}
prim(true) = bool{x|x}
prim(false) = bool{x|!x}
```

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

```
prim(add) = x:int → y:int → int{z|z == x + y}
prim(sub) = x:int → y:int → int{z|z == x - y}
```

and comparison operations as in

```
prim(leq) = x:int → y:int → bool{z|z == x < y}
prim(get) = x:int → y:int → bool{z|z == x >= y}
```

- 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

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

The use of `self` guarantees that we further strengthen base type refinements by bringing 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

```
abs = λx. let c = leq(0, x) in if (c) then (x) else (sub 0 x)
```

of type

```
abs : x:int → 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 $\Gamma \vdash x \Leftarrow t_1$ in the premises of Syn-App. For example, given the context

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

the term **add nat one** would synthesize

$$\Gamma \vdash \text{add nat one} \Rightarrow \text{int}\{z \mid z == \text{nat} + \text{one}\}$$

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

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

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 $\exists x:s.t$ 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 $\lambda x.e$ 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 $x:t_1 \rightarrow t_2$ if its body e can be checked against t_2 with the context extended with the argument binding $x : t_1$ as defined by the rule Chk-Fun.
- Let Binding terms $\text{let } x = e_1 \text{ in } e_2$ 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 t_2 if e_2 can be checked against t_2 with the context extended with the binding $x : t_1$ which has to be synthesized for e_1 .
- Recursive Binding terms $\text{rec } x = e_1:t_1 \text{ in } e_2$ 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 e_1 , and by pushing the type binding $x : t_1$ into the checking of e_1 instead of synthesizing its type.
- If Expression terms $\text{if } x \text{ then } e_1 \text{ else } e_2$ 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 e_1 and 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 e_1 and e_2 (JHALA; VAZOU, 2020). If this binding was not present in the context, the checking relation

$$\begin{aligned} \text{not} &= \lambda x. \text{if } x \text{ then false else true} \\ \Gamma \vdash \text{not} &\Leftarrow x:\text{bool} \rightarrow \text{bool}\{b|b == !x\} \end{aligned}$$

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 subsumption rule Chk-Syn connects the checking and synthesis judgments by defining that if a term e synthesized a type t_1 and the type t_1 is subsumed by type t_2 (i.e., t_1 is subtype of t_2) then we can check e against 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 \mathbf{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 $(x :: t) \implies c$ defined as

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

The subtyping relation can be implemented as a function **sub** that takes two types t_1 and t_2 as input and outputs a constraint c which the validity of c implies the subtyping relation $t_1 \prec t_2$ formalized by Jhala and Vazou (2020) as the following proposition:

$$\text{if } \mathbf{sub}(t_1, t_2) = c \text{ and } \Gamma \vdash c \text{ then } \Gamma \vdash t_1 \prec t_2$$

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.

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

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:

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

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:

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

```

synth( $\Gamma$ ,  $x$ ) = (true, self( $x$ ,  $t$ ))
  where:
     $x : t \in \Gamma$ 
synth( $\Gamma$ ,  $c$ ) = (true, prim( $c$ ))
synth( $\Gamma$ ,  $e \ y$ ) = ( $c_1 \wedge c_2$ ,  $t_2[x := y]$ )
  where:
     $(c_1, x:t_1 \rightarrow t_2) = \text{synth}(\Gamma, e)$ 
     $c_2 = \text{check}(\Gamma, y, t_1)$ 
synth( $\Gamma$ ,  $e:t$ ) = ( $c$ ,  $t$ )
  where:
     $c = \text{check}(\Gamma, e, t)$ 

```

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.

$\text{check}(\Gamma, e, t_2) = c_1 \wedge c_2$

where:

$$(c_1, t_1) = \text{synth}(\Gamma, e)$$

$$c_2 = \text{sub}(t_1, t_2)$$

$\text{check}(\Gamma, \lambda x. e, x : t_1 \rightarrow t_2) = (x :: t_1) \implies c$

where:

$$c = \text{check}(\Gamma_1, e, t)$$

$$\Gamma_1 = \Gamma, x : t_1$$

$\text{check}(\Gamma, \text{let } x = e_1 \text{ in } e_2, t_2) = c_1 \wedge (x :: t_1) \implies c_2$

where:

$$(c_1, t_1) = \text{synth}(\Gamma, e_1)$$

$$c_2 = \text{check}(\Gamma_1, e_2, t_2)$$

$$\Gamma_1 = \Gamma, x : t_1$$

$\text{check}(\Gamma, \text{rec } x = e_1 : t_1 \text{ in } e_2, t_2) = c_1 \wedge c_2$

where:

$$c_1 = \text{check}(\Gamma_1, e_1, t_1)$$

$$c_2 = \text{check}(\Gamma_1, e_2, t_2)$$

$$\Gamma_1 = \Gamma, x : t_1$$

$\text{check}(\Gamma, \text{if } x \text{ then } e_1 \text{ else } e_2, t) = c_1 \wedge c_2$

where:

$$c_1 = (y :: \text{int}\{x|x\}) \implies \text{check}(\Gamma, e_1, t)$$

$$c_2 = (y :: \text{int}\{x|!x\}) \implies \text{check}(\Gamma, e_2, t)$$

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 → ,	Slash → /	
Collon → :	Percent → %	ThinArrow → ->
SemiColon → ;	Greater → >	FatArrow → =>
OpenParenthesis → (Less → <	FnKw → fn
CloseParenthesis →)	Exclamation → !	LetKw → let
OpenBraces → {	Pipe →	IfKw → if
CloseBraces → }	DoubleCollon → ::	ElseKw → else
Equals → =	DoubleEquals → ==	TrueKw → true
Plus → +	ExclamationEquals → !=	FalseKw → false
Minus → -	GreaterEquals → >=	TypeKw → type
Asterisk → *	LessEquals → <=	MatchKw → match
Ampersand → &	DoublePipe →	NewKw → new
	DoubleAmpersand → &&	
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 than 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**:

```

1  type SomeTypeName {
2    SomeConstructorName(Type1, Type2, ... ),
3    ...
4  }
5  fn some_fn_name(arg_name1: Type1, arg_name_2: Type2, ...) -> {
6    ...
7 }
```

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 aims 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 second one was employed to simplify the subsequent translation into LLVM-IR by avoiding the need to create unnamed 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 were taken from module `hir`, submodules `liquid` and `check`, found in Appendix A

```

1 // Base type with a refinement predicate: b{x|p}
2 struct RefinedBase {
3     pub base: Type,
4     pub binder: Name,
5     pub predicate: Predicate,
6 }
7
8 // x:t -> t
9 struct DependentFunction {
10    pub parameter: (Name, RefinedBase),
11    pub tail_type: Box<RefinedType>,
12 }
13
14 // t
15 enum RefinedType {
16     Base(RefinedBase),
17     Fn(DependentFunction),
18 }
19
20 pub enum Type {
21     AbstractDataType(TypeDefinitionId /* internal ID for our AST database */),
22     FunctionDefinition(CallableDefinitionId /* internal ID for our AST database */),
23     Pointer(Box<Type>),
24     Scalar(ScalarType),
25 }
26
27 pub enum ScalarType {
28     Integer(IntegerKind /* I32, I64 */),
29     Boolean,
30 }
31
32 struct Context {
33     bindings: Vec<(Path, RefinedType)>,
34 }
```

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.

```

1  pub enum Predicate {
2      Variable(Name),
3      Boolean(bool),
4      Integer(u128),
5      // contains arithmetic and boolean operators
6      Binary(BinaryOperator, Box<Predicate>, Box<Predicate>),
7      Unary(UnaryOperator, Box<Predicate>),
8  }
9
10 enum Constraint {
11     Predicate(Predicate),
12     Implication {
13         binder: Name,
14         base: Type,
15         antecedent: Predicate,
16         consequent: Box<Self>,
17     },
18     Conjunction(Box<Self>, Box<Self>),
19 }
```

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:

```

1 // returns the type of the term and a constraint that must be satisfied
2 fn synth_type(self, term_id: TermId) -> (Self, RefinedType, Option<Constraint>)
3
4 // checks that the type of the lesser term is a subtype of the greater type
5 fn subtype(self, lesser: RefinedBase, greater: RefinedBase) -> (Self, Constraint)
6
7 // checks that the term has type constraint_type
8 fn check_type(mut self, term_id: TermId, constraint_type: RefinedBase) -> (Self,
9    Constraint) {
10   let term = &self.body.expressions[term_id];
11   match term {
12     Term::Block {
13       statements,
14       trailing_expression,
15     } => {
16       // ... omitted for brevity, but
17       // we check that the trailing expression has type constraint_type
18       // among other things
19     }
20     Term::If {
21       condition,
22       then_branch,
23       else_branch,
24     } => {
25       // ... omitted for brevity, but
26       // we check that condition has type bool
27       // c1, c2 are the constraints for the then and else branches
28       // both must be satisfied
29       (fold, Constraint::Conjunction(c1.into(), c2.into()))
30     }
31     _ => {
32       let (fold, t, c) = self.synth_type(term_id);
33       // we verify that t <: constraint_type
34       let (fold, c2) = fold.subtype(t.as_refined_base(), constraint_type);
35       // both c and c2 must be satisfied
36       (fold, Constraint::make_conjunction(c, c2))
37     }
38   }

```

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.

```

1 // within a context, checks if a constraint can be satisfied
2 fn entailment(context: Context, constraint: Constraint) -> bool {
3     match context.pop() {
4         (_, None) => solve(constraint),
5         (tail, Some((path, RefinedType::Fn(_)))) => entailment(tail, constraint),
6         (tail, Some((path, RefinedType::Base(RefinedBase { binder, predicate, base })))) => {
7             let context_binder = path.as_name();
8             let predicate = substitution(binder, context_binder.clone(), predicate);
9
10            entailment(
11                tail,
12                Constraint::Implication {
13                    binder: context_binder,
14                    base,
15                    antecedent: predicate,
16                    consequent: constraint.into(),
17                },
18            )
19        }
20    }
21 }
22
23 // substitutes all occurrences of old by new in predicate
24 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.

```

1 // gets the constraint, flattens it and then pass it to an SMT solver
2 // to check if the constraint can be satisfied or not
3 fn solve(constraint: Constraint) -> bool
4
5 // flattens a complex constraint into a data structure that the SMT

```

```

6 // solver can process more easily
7 fn flatten(constraint: Constraint) -> Vec<FlatImplicationConstraint>
8
9 struct FlatImplicationConstraint {
10     binders: Vec<(Name, Type)>,
11     antecedent: Predicate,
12     consequent: Predicate,
13 }

```

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.

```

1 fn fold_binary_expression(
2     &self,
3     indirect_value: Option<PointerValue<'context>>,
4     operator: &BinaryOperator,
5     lhs: &TermId,
6     rhs: &TermId,
7 ) -> Option<Value<'context>> {
8     let lhs = self.fold_expression(None, *lhs);
9     let rhs = self.fold_expression(None, *rhs);
10    let int_value = match operator {
11        BinaryOperator::Arithmetic(arithmetc_op) => match arithmetc_op {
12            ArithmeticOperator::Add => self.builder.build_int_add(lhs, rhs, ""),
13            // ...
14        },

```

```

15     BinaryOperator::Compare(compare_op) => {
16         // ...
17     }
18 };
19 match indirect_value {
20     Some(ptr) => {
21         self.builder.build_store(ptr, int_value.as_basic_value_enum());
22         None
23     }
24     None => Some(Value::new(ValueKind::Direct, int_value.into())),
25 }
26 }
```

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.

```

1 fn fold_call_expression(
2     &self,
3     indirect_value: Option<PointerValue<'context>>,
4     callee: &TermId,
5     arguments: &[TermId],
6 ) -> Option<Value<'context>> {
7     let callee_type = &self.inference.type_of_expression[*callee];
8     let callable_definition = match callee_type {
9         Type::FunctionDefinition(callable) => callable,
10        _ => panic!("call has no callable type."),
11    };
12    match callable_definition {
13        CallableDefinitionId::FunctionDefinition(id) => {
14            let function_info = self.function_info_cache.function_info(&id);
15            let function_value = self.function_value_cache.llvm_function_value(&id);
16
17            let return_type = function_info.get_return_type(function_value);
18
19            // Allocates the return value if it is not returned directly via registers
20            // i.e. returned indirectly through memory
21            let indirect_function_return = match function_info.return_kind {
22                ValueKind::Direct => None,
23                ValueKind::Indirect => match indirect_value {
24                    Some(ptr) => Some(ptr.as_basic_value_enum()),
25                    None => Some(
26                        self.get_alloca_builder()
27                            .build_alloca(
28                                BasicType-
29                                    ↳ Enum::try_from(return_type.into_pointer_type().get_element_type()).unwrap(),
30                                ...),
31                            .build(),
32                            ...),
33                }
34            }
35            self.builder.build_call(callee, arguments, indirect_function_return);
36        }
37    }
38 }
```

```

29             """",
30         )
31         .as_basic_value_enum(),
32     ),
33 },
34 };
35 // Creates LLVM values for arguments/parameters whether they are storable
36 // in registers or they are stored in memory (and therefore needs alloc/store
37 // → instructions)
38 let arguments = ...
39
40 // Creating LLVM call instruction
41 let call_value = self.builder.build_call(
42     function_value,
43     arguments.map(|x| x.into()).collect::<Vec<_>>().as_slice(),
44     """",
45 );
46
47 // Returns to the upper level whether the return value is stored in registers
48 match indirect_value {
49     Some(_) => None,
50     None => match function_info.return_kind {
51         ValueKind::Indirect => Some(Value::new(
52             ValueKind::Indirect,
53             indirect_function_return.unwrap(),
54         )),
55         ValueKind::Direct => Some(Value::new(
56             ValueKind::Direct,
57             call_value.try_as_basic_value().unwrap_left(),
58         )),
59     },
60 }
61 CallableDefinitionId::ValueConstructor(constructor_id) => {
62     // Constructs a ADT value.
63 }
64 }
65 }
```

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.

```

1 fn fold_if_expression(
2     &self,
```

```

3   indirect_value: Option<PointerValue<'context>>,
4   condition: &TermId,
5   then_branch: &TermId,
6   else_branch: &TermId,
7 ) -> Option<Value<'context>> {
8   let comparison = self.fold_expression(None, *condition);
9
10  let then_block = self.context.append_basic_block(self.get_owener_function_value(),
11    ↪ "then");
12  let else_block = self.context.append_basic_block(self.get_owener_function_value(),
13    ↪ "else");
14  let merge_block = self.context.append_basic_block(self.get_owener_function_value(),
15    ↪ "merge");
16
17 // Creates branch instruction to then branch
18 self.builder.build_conditional_branch(comparison, then_block, else_block);
19
20 self.builder.position_at_end(then_block);
21 let then_value = self.fold_expression(indirect_value, *then_branch);
22 let then_block = self.builder.get_insert_block().unwrap();
23 // Creates branch instruction to go to the merge block
24 self.builder.build_unconditional_branch(merge_block);
25
26 self.builder.position_at_end(else_block);
27 let else_value = self.fold_expression(indirect_value, *else_branch);
28 let else_block = self.builder.get_insert_block().unwrap();
29 // Creates branch instruction to go to the merge block
30 self.builder.build_unconditional_branch(merge_block);
31
32 match indirect_value {
33   Some(ptr) => None,
34   None => {
35     let (
36       Value { kind: then_kind, value: then_value },
37       Value { kind: else_kind, value: else_value },
38     ) = (then_value.unwrap(), else_value.unwrap());
39     match (then_kind, else_kind) {
40       (ValueKind::Direct, ValueKind::Direct) | (ValueKind::Indirect,
41         ↪ ValueKind::Indirect) => {
42       let phi = self.builder.build_phi(then_value.get_type(), "phi");
43       phi.add_incoming(&[(&then_value, then_block), (&else_value, else_block)]);
44       Some(Value::new(then_kind, phi.as_basic_value()))
45     }
46   }
47 }

```

```
47     }
48 }
```

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.

```
1 pub struct CodeGenTypeCache<'db, 'context> {
2   db: &'db dyn CodeGenDatabase,
3   context: &'context Context,
4   target_data: TargetData,
5   adt_map: RefCell<HashMap<TypeID, (StructType<'context>,
6     ↵ TypeInfo<'context>) >>,
7   adt_variant_map: RefCell<HashMap<ValueConstructorID, StructType<'context>>>,
8 }
```

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.

```

1 struct CodeGenFunctionInfoCache<'db, 'context, 'type_cache> {
2   db: &'db dyn CodeGenDatabase,
3   context: &'context Context,
4   type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
5 }
```

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:

```

1 fn id(x: {y: i64 | true}) -> {z: i64 | z == x} {
2   x
3 }
```

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

```

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

```

5     Name { id: "x" },
6     RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" }, predicate:
7       → Boolean(true) }
8   ),
9   tail_type: Base(RefinedBase {
10     base: Scalar(Integer(I64)), binder: Name { id: "z" },
11     predicate: Binary(
12       Compare(Equality { negated: false }), Variable(Name { id: "z" }),
13       → Variable(Name { id: "x" })
14     )
15   ),
16   ( Path { segments: [Name { id: "x" }] },
17     Base(RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" },
18       → predicate: Boolean(true) })
19   ),
20 ] }
```

The flattened constraint generate is:

```

1 Context { bindings: [
2   ( Path { segments: [Name { id: "id" }] },
3     Fn(DependentFunction {
4       parameter: (
5         Name { id: "x" },
6         RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" }, predicate:
7           → Boolean(true) }
8       ),
9       tail_type: Base(RefinedBase {
10         base: Scalar(Integer(I64)), binder: Name { id: "z" },
11         predicate: Binary(
12           Compare(Equality { negated: false }), Variable(Name { id: "z" }),
13           → Variable(Name { id: "x" })
14         )
15       )
16     ),
17     ( Path { segments: [Name { id: "x" }] },
18       Base(RefinedBase { base: Scalar(Integer(I64)), binder: Name { id: "y" },
19         → predicate: Boolean(true) })
20   ),
21 ] }
```

After translation to Z3:

```

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

```

3 (assert (and true true (= y x) true true (not (= y x))))
4
5 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:

```

1 ; ModuleID = 'ekitai_module'
2 source_filename = "ekitai_module"
3 target datalayout =
4   → "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
5 target triple = "x86_64-pc-linux-gnu"
6
6 define i64 @id(i64 %x) {
7   ret i64 %x
8 }
```

5.5.2 Absolute value function with refinements

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

```

1 fn abs_liquid(x: {y: i64 | true}) -> {z: i64 | z >= 0} {
2   if x > 0 {
3     x
4   } else {
5     -x
6   }
7 }
```

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

```

1 Context: Context {
2   bindings: [
3     (
4       Path {
5         segments: [
6           Name {
7             id: "abs_liquid",
8           },
9         ],
10      },
11      Fn(
12        DependentFunction {
```

```

13      parameter: (
14          Name {
15              id: "x",
16          },
17          RefinedBase {
18              base: Scalar(
19                  Integer(
20                      I64,
21                  ),
22              ),
23              binder: Name {
24                  id: "y",
25              },
26              predicate: Boolean(
27                  true,
28              ),
29          },
30      ),
31      tail_type: Base(
32          RefinedBase {
33              base: Scalar(
34                  Integer(
35                      I64,
36                  ),
37              ),
38              binder: Name {
39                  id: "z",
40              },
41              predicate: Binary(
42                  Compare(
43                      Order {
44                          ordering: Greater,
45                          strict: false,
46                      },
47                  ),
48                  Variable(
49                      Name {
50                          id: "z",
51                      },
52                  ),
53                  Integer(
54                      0,
55                  ),
56                  ),
57              },
58          ),
59      },
60  ),

```

```

61     ),
62     (
63         Path {
64             segments: [
65                 Name {
66                     id: "x",
67                 },
68             ],
69         },
70     Base(
71         RefinedBase {
72             base: Scalar(
73                 Integer(
74                     I64,
75                 ),
76             ),
77             binder: Name {
78                 id: "y",
79             },
80             predicate: Boolean(
81                 true,
82             ),
83         },
84     ),
85     ),
86     (
87         Path {
88             segments: [
89                 Name {
90                     id: "__arg0",
91                 },
92             ],
93         },
94     Base(
95         RefinedBase {
96             base: Scalar(
97                 Integer(
98                     I64,
99                 ),
100            ),
101            binder: Name {
102                id: "y",
103            },
104            predicate: Binary(
105                Logic(
106                    And,
107                ),
108                Boolean(

```

```

109                     true,
110                 ),
111                 Binary(
112                     Compare(
113                         Equality {
114                             negated: false,
115                         },
116                     ),
117                     Variable(
118                         Name {
119                             id: "y",
120                         },
121                     ),
122                     Variable(
123                         Name {
124                             id: "x",
125                         },
126                     ),
127                     ),
128                 ),
129             },
130         ),
131     ),
132     (
133         Path {
134             segments: [
135                 Name {
136                     id: "__arg1",
137                     },
138                 ],
139             },
140             Base(
141                 RefinedBase {
142                     base: Scalar(
143                         Integer(
144                             I64,
145                         ),
146                     ),
147                     binder: Name {
148                         id: "lit",
149                     },
150                     predicate: Binary(
151                         Compare(
152                             Equality {
153                                 negated: false,
154                                 },
155                             ),
156                             Variable(

```

```

157             Name {
158                 id: "lit",
159             },
160         ),
161         Integer(
162             0,
163         ),
164     ),
165     },
166 ),
167 ),
168 (
169     Path {
170         segments: [
171             Name {
172                 id: "__arg2",
173             },
174         ],
175     },
176     Base(
177         RefinedBase {
178             base: Scalar(
179                 Boolean,
180             ),
181             binder: Name {
182                 id: "ret",
183             },
184             predicate: Binary(
185                 Compare(
186                     Equality {
187                         negated: false,
188                     },
189                 ),
190                 Variable(
191                     Name {
192                         id: "ret",
193                     },
194                 ),
195                 Binary(
196                     Compare(
197                         Order {
198                             ordering: Greater,
199                             strict: true,
200                         },
201                     ),
202                     Variable(
203                         Name {
204                             id: "__arg0",

```



```

253             Variable(
254                 Name {
255                     id: "x",
256                 },
257             ),
258         ),
259     ],
260     ],
261     ),
262     ],
263 ],
264 }
```

Followed by the flattened constraint generated from the context:

```

1 Constraint: Implication {
2     binder: Name {
3         id: "x",
4     },
5     base: Scalar(
6         Integer(
7             I64,
8         ),
9     ),
10    antecedent: Boolean(
11        true,
12    ),
13    consequent: Conjunction(
14        Implication {
15            binder: Name {
16                id: "__arg0",
17            },
18            base: Scalar(
19                Integer(
20                    I64,
21                ),
22            ),
23            antecedent: Binary(
24                Logic(
25                    And,
26                ),
27                Boolean(
28                    true,
29                ),
30                Binary(
31                    Compare(
32                        Equality {
33                            negated: false,
```

```

34             },
35         ),
36         Variable(
37             Name {
38                 id: "__arg0",
39             },
40         ),
41         Variable(
42             Name {
43                 id: "x",
44             },
45         ),
46     ),
47 ),
48 consequent: Implication {
49     binder: Name {
50         id: "__arg1",
51     },
52     base: Scalar(
53         Integer(
54             I64,
55         ),
56     ),
57     antecedent: Binary(
58         Compare(
59             Equality {
60                 negated: false,
61             },
62         ),
63         Variable(
64             Name {
65                 id: "__arg1",
66             },
67         ),
68         Integer(
69             0,
70         ),
71     ),
72     consequent: Conjunction(
73         Implication {
74             binder: Name {
75                 id: "y",
76             },
77             base: Scalar(
78                 Integer(
79                     I64,
80                 ),
81             ),

```

```

82      antecedent: Binary(  

83          Logic(  

84              And,  

85          ),  

86          Binary(  

87              Logic(  

88                  And,  

89          ),  

90          Boolean(  

91              true,  

92          ),  

93          Binary(  

94              Compare(  

95                  Equality {  

96                      negated: false,  

97                  },  

98          ),  

99          Variable(  

100             Name {  

101                 id: "y",  

102             },  

103         ),  

104         Variable(  

105             Name {  

106                 id: "x",  

107             },  

108         ),  

109     ),  

110 ),  

111     Binary(  

112         Compare(  

113             Equality {  

114                 negated: false,  

115             },  

116         ),  

117         Variable(  

118             Name {  

119                 id: "y",  

120             },  

121         ),  

122         Variable(  

123             Name {  

124                 id: "__arg0",  

125             },  

126         ),  

127     ),  

128 ),  

129 consequent: Predicate(  


```



```

226   ),
227   consequent: Conjunction(
228     Implication {
229       binder: Name {
230         id: "__fresh0",  

231       },
232       base: Scalar(  

233         Boolean,  

234       ),  

235       antecedent: Variable(  

236         Name {  

237           id: "__arg2",  

238         },  

239       ),  

240       consequent: Implication {  

241         binder: Name {  

242           id: "y",  

243         },  

244         base: Scalar(  

245           Integer(  

246             I64,  

247           ),  

248         ),  

249         antecedent: Binary(  

250           Logic(  

251             And,  

252           ),  

253             Boolean(  

254               true,  

255             ),  

256             Binary(  

257               Compare(  

258                 Equality {  

259                   negated: false,  

260                 },  

261               ),  

262               Variable(  

263                 Name {  

264                   id: "y",  

265                 },  

266               ),  

267               Variable(  

268                 Name {  

269                   id: "x",  

270                 },  

271               ),  

272             ),  

273           ),

```

```

274     consequent: Predicate(  

275         Binary(  

276             Compare(  

277                 Order {  

278                     ordering: Greater,  

279                     strict: false,  

280                 },  

281             ),  

282             Variable(  

283                 Name {  

284                     id: "y",  

285                 },  

286             ),  

287             Integer(  

288                 0,  

289             ),  

290             ),  

291         ),  

292     ),  

293 },  

294 Implication {  

295     binder: Name {  

296         id: "__fresh0",  

297     },  

298     base: Scalar(  

299         Boolean,  

300     ),  

301     antecedent: Unary(  

302         Negation,  

303         Variable(  

304             Name {  

305                 id: "__arg2",  

306             },  

307         ),  

308     ),  

309     consequent: Implication {  

310         binder: Name {  

311             id: "__arg3",  

312         },  

313         base: Scalar(  

314             Integer(  

315                 I64,  

316             ),  

317         ),  

318         antecedent: Binary(  

319             Logic(  

320                 And,  

321             ),  

322         ),  

323     ),  

324     consequent: Predicate(  

325         Binary(  

326             Compare(  

327                 Order {  

328                     ordering: Greater,  

329                     strict: false,  

330                 },  

331             ),  

332             Variable(  

333                 Name {  

334                     id: "y",  

335                 },  

336             ),  

337             Integer(  

338                 0,  

339             ),  

340             ),  

341         ),  

342     ),  

343 
```

```

322     Boolean(
323         true,
324     ),
325     Binary(
326         Compare(
327             Equality {
328                 negated: false,
329             },
330         ),
331         Variable(
332             Name {
333                 id: "__arg3",
334             },
335         ),
336         Variable(
337             Name {
338                 id: "x",
339             },
340         ),
341         ),
342     ),
343     consequent: Conjunction(
344         Implication {
345             binder: Name {
346                 id: "y",
347             },
348             base: Scalar(
349                 Integer(
350                     I64,
351                 ),
352             ),
353             antecedent: Binary(
354                 Logic(
355                     And,
356                 ),
357                 Binary(
358                     Logic(
359                         And,
360                     ),
361                     Boolean(
362                         true,
363                     ),
364                     Binary(
365                         Compare(
366                             Equality {
367                                 negated: false,
368                             },
369                         ),
370                     ),
371                 ),
372             ),
373         )
374     )
375 
```

```

370             Variable(
371                 Name {
372                     id: "y",
373                 },
374             ),
375             Variable(
376                 Name {
377                     id: "x",
378                 },
379             ),
380             ),
381         ),
382         Binary(
383             Compare(
384                 Equality {
385                     negated: false,
386                 },
387             ),
388             Variable(
389                 Name {
390                     id: "y",
391                 },
392             ),
393             Variable(
394                 Name {
395                     id: "__arg3",
396                 },
397             ),
398             ),
399         ),
400         consequent: Predicate(
401             Boolean(
402                 true,
403             ),
404             ),
405         },
406         Implication {
407             binder: Name {
408                 id: "ret",
409             },
410             base: Scalar(
411                 Integer(
412                     I64,
413                 ),
414             ),
415             antecedent: Binary(
416                 Compare(
417                     Equality {

```

```

418                     negated: false,
419                     },
420                     ),
421                     Variable(
422                         Name {
423                             id: "ret",
424                         },
425                     ),
426                     Unary(
427                         Minus,
428                         Variable(
429                             Name {
430                                 id: "__arg3",
431                             },
432                         ),
433                     ),
434                     ),
435                     consequent: Predicate(
436                         Binary(
437                             Compare(
438                                 Order {
439                                     ordering: Greater,
440                                     strict: false,
441                                     },
442                                 ),
443                                 Variable(
444                                     Name {
445                                         id: "ret",
446                                         },
447                                     ),
448                                     Integer(
449                                         0,
450                                         ),
451                                     ),
452                                     ),
453                                     },
454                                     ),
455                                     },
456                                     },
457                                     ),
458                                     },
459                                     ),
460                     }

```

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:

1 **Solver**:

```

2  (declare-fun x () Int)
3  (declare-fun __arg0 () Int)
4  (declare-fun __arg1 () Int)
5  (declare-fun __arg2 () Bool)
6  (declare-fun ret () Bool)
7  (assert (and true
8      (= ret (> __arg0 __arg1))
9      (= ret __arg2)
10     (= __arg2 (> __arg0 __arg1)))
11     (= __arg1 0)
12     true
13     (= __arg0 x)
14     true
15     (not true)))
16
17 Result: Unsat
18 Solver:
19 (declare-fun x () Int)
20 (declare-fun __arg0 () Int)
21 (declare-fun __arg1 () Int)
22 (declare-fun __arg2 () Bool)
23 (declare-fun __arg3 () Int)
24 (declare-fun y () Int)
25 (assert (and true
26     true
27     (= y x)
28     (= y __arg0)
29     (= __arg1 0)
30     (and true (= __arg0 x)))
31     true
32     true
33     (= __arg3 x)
34     (= __arg2 (> __arg0 __arg1))
35     (= __arg1 0)
36     (and true (= __arg0 x)))
37     true
38     (not true)))
39
40 Result: Unsat
41 Solver:
42 (declare-fun x () Int)
43 (declare-fun __arg0 () Int)
44 (declare-fun __arg1 () Int)
45 (declare-fun __arg2 () Bool)
46 (declare-fun __arg3 () Int)
47 (declare-fun lit () Int)
48 (assert (and true
49     (= lit 0))

```

```

50      (= lit __arg1)
51      (= __arg1 0)
52      (and true (= __arg0 x))
53      true
54      true
55      (= __arg3 x)
56      (= __arg2 (> __arg0 __arg1))
57      (= __arg1 0)
58      (and true (= __arg0 x))
59      true
60      (not true)))
61
62 Result: Unsat
63 Solver:
64 (declare-fun y () Int)
65 (declare-fun x () Int)
66 (declare-fun __arg0 () Int)
67 (declare-fun __arg1 () Int)
68 (declare-fun __arg2 () Bool)
69 (declare-fun __arg3 () Int)
70 (assert (and true
71           true
72           (= y x)
73           __arg2
74           (= __arg2 (> __arg0 __arg1))
75           true
76           true
77           (= __arg3 x)
78           (= __arg2 (> __arg0 __arg1))
79           (= __arg1 0)
80           true
81           (= __arg0 x)
82           true
83           (not (>= y 0))))
84
85 Result: Unsat
86 Solver:
87 (declare-fun x () Int)
88 (declare-fun __arg0 () Int)
89 (declare-fun __arg1 () Int)
90 (declare-fun __arg2 () Bool)
91 (declare-fun __arg3 () Int)
92 (declare-fun y () Int)
93 (assert (and true
94           true
95           (= y x)
96           (= y __arg3)
97           (and true (= __arg3 x)))

```

```

98      (not __arg2)
99      (= __arg2 (> __arg0 __arg1))
100     true
101     (and true (= __arg3 x))
102     (= __arg2 (> __arg0 __arg1))
103     (= __arg1 0)
104     true
105     (= __arg0 x)
106     true
107     (not true)))
108
109 Result: Unsat
110 Solver:
111 (declare-fun ret () Int)
112 (declare-fun x () Int)
113 (declare-fun __arg0 () Int)
114 (declare-fun __arg1 () Int)
115 (declare-fun __arg2 () Bool)
116 (declare-fun __arg3 () Int)
117 (assert (and true
118           (= ret (- __arg3)))
119           (and true (= __arg3 x)))
120           (not __arg2)
121           (= __arg2 (> __arg0 __arg1))
122           true
123           (and true (= __arg3 x))
124           (= __arg2 (> __arg0 __arg1))
125           (= __arg1 0)
126           true
127           (= __arg0 x)
128           true
129           (not (>= ret 0))))
130
131 Result: Unsat

```

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

```

1 ; ModuleID = 'ekitai_module'
2 source_filename = "ekitai_module"
3 targetdatalayout =
4   → "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
4 target triple = "x86_64-pc-linux-gnu"
5
6 define i64 @abs_liquid(i64 %x) {
7   %l = icmp sgt i64 %x, 0
8   br i1 %l, label %then, label %else
9

```

```
10 then: ; preds = %0
11     br label %merge
12
13 else: ; preds = %0
14     %2 = sub i64 0, %x
15     br label %merge
16
17 merge: ; preds = %else, %then
18     %phi = phi i64 [ %x, %then ], [ %2, %else ]
19     ret i64 %phi
20 }
```

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 annexed to this document. Please refer to the code's repository hosted at <https://github.com/tarberd/ekitai/>.

A.1 SUBMODULE HIR::LIQUID

```

1  use core::panic;
2  use std::vec;
3
4  use la_arena::Idx;
5  use z3::ast::{Ast, Bool, Int};
6
7  use crate::{
8      check::{
9          type_inference::{InferenceResult, TypeReferenceResolver},
10         IntegerKind, ScalarType, Type,
11     },
12     semantic_ir::{
13         definition_map::{FunctionDefinitionData, FunctionDefinitionId},
14         intrinsic::BuiltinInteger,
15         name::Name,
16         path::Path,
17         path_resolver::Resolver,
18         refinement::{Predicate, UnaryOperator},
19         term::{
20             ArithmeticOperator, BinaryOperator, Body, CompareOperator, Literal,
21             ↳ LogicOperator,
22             Pattern, PatternId, Statement, Term, TermId, UnaryOperator as TermUnaryOp,
23         },
24         type_reference::TypeReference,
25     },
26     HirDatabase,
27 };
28 #[derive(Debug, Clone)]
29 struct RefinedBase {
30     pub base: Type,
31     pub binder: Name,
32     pub predicate: Predicate,
33 }
34
35 #[derive(Debug, Clone)]
36 struct DependentFunction {
37     pub parameter: (Name, RefinedBase),

```

```

38     pub tail_type: Box<RefinedType>,
39 }
40
41 #[derive(Debug, Clone)]
42 enum RefinedType {
43     Base(RefinedBase),
44     Fn(DependentFunction),
45 }
46
47 impl RefinedType {
48     fn as_refined_base(self) -> RefinedBase {
49         match self {
50             RefinedType::Base(base) => base,
51             _ => panic!("Not a RefinedBase."),
52         }
53     }
54
55     fn as_dependent_function(self) -> DependentFunction {
56         match self {
57             RefinedType::Fn(func) => func,
58             _ => panic!("Not a DependentFunction."),
59         }
60     }
61 }
62
63 impl From<RefinedBase> for RefinedType {
64     fn from(base: RefinedBase) -> Self {
65         Self::Base(base)
66     }
67 }
68
69 impl From<DependentFunction> for RefinedType {
70     fn from(fun: DependentFunction) -> Self {
71         Self::Fn(fun)
72     }
73 }
74
75 #[derive(Debug, Clone)]
76 struct Context {
77     bindings: Vec<(Path, RefinedType)>,
78 }
79
80 impl Context {
81     fn get(&self, path: &Path) -> Option<RefinedType> {
82         self.bindings
83             .iter()
84             .find(|(to_find, _)| path == to_find)
85             .map(|(_, ty)| ty.clone())

```

```

86     }
87
88     fn pop(mut self) -> (Self, Option<(Path, RefinedType)>) {
89         let opt = self.bindings.pop();
90         (self, opt)
91     }
92 }
93
94 impl FromIterator<(Path, RefinedType)> for Context {
95     fn from_iter<T: IntoIterator<Item = (Path, RefinedType)>>(iter: T) -> Self {
96         let bindings = iter.into_iter().collect();
97         Self { bindings }
98     }
99 }
100
101 #[derive(Debug)]
102 enum Constraint {
103     Predicate(Predicate),
104     Implication {
105         binder: Name,
106         base: Type,
107         antecedent: Predicate,
108         consequent: Box<Self>,
109     },
110     Conjunction(Box<Self>, Box<Self>),
111 }
112
113 impl Constraint {
114     fn make_conjunction(left: Option<Self>, right: Self) -> Self {
115         match left {
116             Some(left) => Self::Conjunction(left.into(), right.into()),
117             None => right,
118         }
119     }
120 }
121
122 fn something(
123     db: &dyn HirDatabase,
124     resolver: &Resolver,
125     type_reference: &TypeReference,
126 ) -> RefinedBase {
127     match type_reference {
128         TypeReference::Path(_) => todo!(),
129         TypeReference::Refinement(inner, binder, predicate) => {
130             let type_ref_resolver = TypeReferenceResolver::new(db, resolver);
131             let inner_type = type_ref_resolver.resolve_type_reference(&inner).unwrap();
132             RefinedBase {
133                 base: inner_type,

```

```

134         binder: binder.clone(),
135         predicate: predicate.clone(),
136     }
137 }
138 TypeReference::Pointer(_) => todo!(),
139 }
140 }
141
142 pub fn check_abstraction(db: &dyn HirDatabase, function_id: FunctionDefinitionId) ->
143     bool {
144     let context = {
145         db.source_file_definitions_map()
146             .root_module_item_scope()
147             .iter_function_locations()
148             .map(|fid| {
149                 let FunctionDefinitionData { name, .. } =
150                     db.function_definition_data(*fid);
151                 (
152                     Path {
153                         segments: vec![name],
154                     },
155                     make_function_type(db, *fid),
156                 )
157             })
158             .collect()
159     };
160
161     let function_type = make_function_type(db, function_id);
162     let body = db.body_of_definition(function_id);
163
164     let (context, constraint) = match function_type {
165         RefinedType::Base(base) => {
166             let (Fold { context, .. }, constraint) = Fold {
167                 body: &body,
168                 context,
169                 inference: db.infer_body_expression_types(function_id),
170                 fresh_var_counter: 0,
171             }
172             .check_type(body.root_expression, base);
173             (context, constraint)
174         }
175         RefinedType::Fn(deffn) => {
176             let (Fold { context, .. }, constraint) = Fold {
177                 body: &body,
178                 context,
179                 inference: db.infer_body_expression_types(function_id),
180                 fresh_var_counter: 0,
181             }

```

```

180         .check_abstraction_type(deffn, body.root_expression);
181         (context, constraint)
182     }
183 };
184
185     println!("Context: {:?}", context);
186     println!("Constraint: {:?}", constraint);
187     entailment(context, constraint)
188 }
189
190 fn make_function_type(db: &dyn HirDatabase, function_id: FunctionDefinitionId) ->
191     RefinedType {
192     let resolver = Resolver::new_for_function(db.upcast(), function_id);
193     let function = db.function_definition_data(function_id);
194     let output_type = something(db, &resolver, &function.return_type);
195
196     let body = db.body_of_definition(function_id.into());
197     let param_types = function
198         .parameter_types
199         .iter()
200         .map(|reference| something(db, &resolver, reference));
201
202     let function_type = body
203         .parameters
204         .iter()
205         .map(|id| {
206             let pat = &body.patterns[*id];
207             match pat {
208                 Pattern::Deconstructor(_, _) => panic!("no deconstructor on parameter"),
209                 Pattern::Bind(name) => name,
210             }
211         })
212         .cloned()
213         .zip(param_types.clone())
214         .rfold(
215             RefinedType::Base(output_type),
216             |tail_type, (param_name, param_type)| {
217                 DependentFunction {
218                     parameter: (param_name, param_type),
219                     tail_type: tail_type.into(),
220                 }
221             }
222         ),
223     );
224     function_type
225 }
226

```

```

227 fn entailment(context: Context, constraint: Constraint) -> bool {
228     match context.pop() {
229         (_, None) => solve(constraint),
230         (tail, Some((_, RefinedType::Fn(_)))) => entailment(tail, constraint),
231         (tail, Some((path, RefinedType::Base(refinement)))) => {
232             let RefinedBase {
233                 binder,
234                 predicate,
235                 base,
236             } = refinement;
237
238             let context_binder = path.as_name();
239             let predicate = substitution(binder, context_binder.clone(), predicate);
240
241             entailment(
242                 tail,
243                 Constraint::Implication {
244                     binder: context_binder,
245                     base,
246                     antecedent: predicate,
247                     consequent: constraint.into(),
248                 },
249             )
250         }
251     }
252 }
253
254 #[derive(Debug, Clone)]
255 enum Z3Predicate<'ctx> {
256     Bool(Bool<'ctx>),
257     Int(Int<'ctx>),
258 }
259
260 impl<'ctx> From<Bool<'ctx>> for Z3Predicate<'ctx> {
261     fn from(from: Bool<'ctx>) -> Self {
262         Z3Predicate::Bool(from)
263     }
264 }
265
266 impl<'ctx> From<Int<'ctx>> for Z3Predicate<'ctx> {
267     fn from(from: Int<'ctx>) -> Self {
268         Z3Predicate::Int(from)
269     }
270 }
271
272 fn solve(constraint: Constraint) -> bool {
273     let flattened_constraint = flatten(constraint);
274 }
```

```

275     let solver_config = z3::Config::new();
276     let solver_context = z3::Context::new(&solver_config);
277     let solver = z3::Solver::new(&solver_context);
278
279     for constraint in flattened_constraint {
280         let constraint = lower(&solver_context, constraint);
281
282         solver.push();
283         solver.assert(&constraint);
284         println!("Solver:\n{solver}");
285         let result = solver.check();
286         println!("Result: {result:?}");
287         solver.pop(1);
288
289         match result {
290             z3::SatResult::Unsat => continue,
291             z3::SatResult::Unknown => return false,
292             z3::SatResult::Sat => return true,
293         };
294     }
295
296     true
297 }
298
299 fn lower(context: &z3::Context, constraint: FlatImplicationConstraint) -> Bool {
300     let FlatImplicationConstraint {
301         binders,
302         antecedent,
303         consequent,
304     } = constraint;
305
306     let variables: Vec<(Name, Z3Predicate)> = binders
307         .into_iter()
308         .map(|(name, base)| match base {
309             Type::AbstractDataType(_) => todo!("no abstract data types supported in
310                 ↪ liquid terms"),
311             Type::FunctionDefinition(_) => todo!("no function data types in liquid
312                 ↪ terms"),
313             Type::Pointer(_) => todo!("no pointer type in liquid terms"),
314             Type::Scalar(scalar) => match scalar {
315                 crate::check::ScalarType::Integer(_) => (
316                     name.clone(),
317                     Int::new_const(context, name.id.as_str()).into(),
318                 ),
319                 crate::check::ScalarType::Boolean => (
320                     name.clone(),
321                     Bool::new_const(context, name.id.as_str()).into(),
322                 ),
323             },
324         })
325         .collect();
326
327     let mut antecedent = context.mk_and();
328     for variable in variables {
329         antecedent = antecedent.add(consequent);
330     }
331
332     let consequent = context.mk_and();
333     for variable in variables {
334         consequent = consequent.add(base);
335     }
336
337     let implication = context.mk_imply(antecedent, consequent);
338
339     if !implication.is_true() {
340         return false;
341     }
342
343     true
344 }
```

```

321         },
322     })
323     .collect();
324
325     let antecedent = lower_predicate(context, &variables, antecedent);
326     let consequent = lower_predicate(context, &variables, consequent);
327
328     match (antecedent, consequent) {
329         (Z3Predicate::Bool(prepo), Z3Predicate::Bool(conse)) => {
330             Bool::and(context, [&prepo, &conse.not()])
331         }
332         _ => todo!(),
333     }
334 }
335
336 fn lower_predicate<'ctx>(
337     context: &'ctx z3::Context,
338     variables: &Vec<(Name, Z3Predicate<'ctx>)>,
339     predicate: Predicate,
340 ) -> Z3Predicate<'ctx> {
341     match predicate {
342         Predicate::Variable(name) => {
343             let x = variables
344                 .iter()
345                 .find(|(to_find, _)| name == *to_find)
346                 .map(|(_, ty)| ty)
347                 .expect("liquid variable not in context");
348             x.clone()
349         }
350         Predicate::Boolean(value) => Bool::from_bool(context, value).into(),
351         Predicate::Integer(value) => Int::from_u64(context, value as u64).into(),
352         Predicate::Binary(op, lhs, rhs) => {
353             let lhs = lower_predicate(context, variables, *lhs);
354             let rhs = lower_predicate(context, variables, *rhs);
355             match op {
356                 BinaryOperator::Arithmetic(arith_op) => {
357                     let (lhs, rhs) = match (lhs, rhs) {
358                         (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => (lhs, rhs),
359                         _ => panic!(),
360                     };
361                     match arith_op {
362                         ArithmeticOperator::Add => Int::add(context, [&lhs, &rhs]),
363                         ArithmeticOperator::Sub => Int::sub(context, [&lhs, &rhs]),
364                         ArithmeticOperator::Div => lhs.div(&rhs),
365                         ArithmeticOperator::Mul => Int::mul(context, [&lhs, &rhs]),
366                         ArithmeticOperator::Rem => lhs.rem(&rhs),
367                     }
368                 .into()

```

```

369 }
370 BinaryOperator::Logic(op) => {
371   let (lhs, rhs) = match (lhs, rhs) {
372     (Z3Predicate::Bool(lhs), Z3Predicate::Bool(rhs)) => (lhs, rhs),
373     _ => todo!(),
374   };
375   match op {
376     LogicOperator::And => Bool::and(context, &[&lhs, &rhs]).into(),
377     LogicOperator::Or => Bool::or(context, &[&lhs, &rhs]).into(),
378   }
379 }
380 BinaryOperator::Compare(compare) => match compare {
381   CompareOperator::Equality { negated } => match (lhs, rhs) {
382     (Z3Predicate::Bool(lhs), Z3Predicate::Bool(rhs)) => match
383       negated {
384         false => lhs._eq(&rhs).into(),
385         true => Bool::distinct(context, &[&lhs, &rhs]).into(),
386       },
387     (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => match negated {
388       false => lhs._eq(&rhs).into(),
389       true => Int::distinct(context, &[&lhs, &rhs]).into(),
390     },
391     _ => panic!("mismatch types in z3"),
392   },
393   CompareOperator::Order { ordering, strict } => match ordering {
394     crate::semantic_ir::term::Ordering::Less => match (lhs, rhs) {
395       (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => match
396         strict {
397           true => lhs.lt(&rhs).into(),
398           false => lhs.le(&rhs).into(),
399         },
400         _ => todo!(),
401     },
402     crate::semantic_ir::term::Ordering::Greater => match (lhs, rhs) {
403       (Z3Predicate::Int(lhs), Z3Predicate::Int(rhs)) => match
404         strict {
405           true => lhs.gt(&rhs).into(),
406           false => lhs.ge(&rhs).into(),
407         },
408         _ => todo!(),
409     },
410   },
411   Predicate::Unary(op, predicate) => {
412     let predicate = lower_predicate(context, variables, *predicate);
413     match (op, predicate) {

```

```

414             (UnaryOperator::Minus, Z3Predicate::Int(int)) =>
415                 int.unary_minus().into(),
416             (UnaryOperator::Negation, Z3Predicate::Bool(boolean)) =>
417                 boolean.not().into(),
418             (UnaryOperator::Minus, Z3Predicate::Bool(_)) => todo!(),
419             (UnaryOperator::Negation, Z3Predicate::Int(_)) => todo!(),
420         }
421     }
422
423     struct FlatImplicationConstraint {
424         binders: Vec<(Name, Type)>,
425         antecedent: Predicate,
426         consequent: Predicate,
427     }
428
429     fn flatten(constraint: Constraint) -> Vec<FlatImplicationConstraint> {
430         flatten_fold(Vec::new(), constraint)
431     }
432
433     fn flatten_fold(
434         mut flattened: Vec<FlatImplicationConstraint>,
435         constraint: Constraint,
436     ) -> Vec<FlatImplicationConstraint> {
437         match constraint {
438             Constraint::Predicate(predicate) => {
439                 flattened.push(FlatImplicationConstraint {
440                     binders: vec![],
441                     antecedent: Predicate::Boolean(true),
442                     consequent: predicate,
443                 });
444                 flattened
445             }
446             Constraint::Implication {
447                 binder,
448                 base,
449                 antecedent,
450                 consequent,
451             } => {
452                 flattened.extend(flatten(*consequent).into_iter().map(
453                     |FlatImplicationConstraint {
454                         mut binders,
455                         antecedent: sub_antecedent,
456                         consequent,
457                     }| {
458                         let binders = {
459                             binders.push((binder.clone(), base.clone()));

```

```

460             binders
461         };
462         let antecedent = Predicate::Binary(
463             BinaryOperator::Logic(LogicOperator::And),
464             sub_antecedent.into(),
465             antecedent.clone().into(),
466         );
467         FlatImplicationConstraint {
468             binders,
469             antecedent,
470             consequent,
471         }
472     },
473 );
474 flattened
475 }
476 Constraint::Conjunction(first, second) => {
477     let flattened = flatten_fold(flattened, *first);
478     flatten_fold(flattened, *second)
479 }
480 }
481 }
482
483 struct Fold<'a> {
484     body: &'a Body,
485     context: Context,
486     inference: InferenceResult,
487     fresh_var_counter: usize,
488 }
489
490 impl<'a> Fold<'a> {
491     fn subtype(self, lesser: RefinedBase, greater: RefinedBase) -> (Self, Constraint) {
492         let RefinedBase {
493             base: lesser_base,
494             binder: lesser_binder,
495             predicate: lesser_predicate,
496         } = lesser;
497         let RefinedBase {
498             base: greater_base,
499             binder: greater_binder,
500             predicate: greater_predicate,
501         } = greater;
502
503         if lesser_base != greater_base {
504             panic!("mismatch base types: {lesser_base:?} <: {greater_base:?}")
505         }
506
507         let constraint = Constraint::Implication {

```

```

508     binder: lesser_binder.clone(),
509     base: lesser_base,
510     antecedent: lesser_predicate,
511     consequent: Constraint::Predicate(substitution(
512         greater_binder,
513         lesser_binder,
514         greater_predicate,
515     ))
516     .into(),
517 };
518
519     (self, constraint)
520 }
521
522 fn check_abstraction_type(
523     mut self,
524     constraint_type: DependentFunction,
525     body_term: TermId,
526 ) -> (Self, Constraint) {
527     let DependentFunction {
528         parameter: (arg_name, arg_ty),
529         tail_type,
530     } = constraint_type;
531     //add to context
532     let arg_path = Path {
533         segments: vec![arg_name.clone()],
534     };
535     self.context
536         .bindings
537         .push((arg_path, arg_ty.clone().into()));
538     //check
539     let (fold, constraint) = match *tail_type {
540         RefinedType::Base(base) => self.check_type(body_term, base),
541         RefinedType::Fn(deffn) => self.check_abstraction_type(deffn, body_term),
542     };
543     //implication constraint
544     let constraint = implication_constraint(arg_name, arg_ty, Some(constraint));
545     (fold, constraint)
546 }
547
548 fn check_type(mut self, term_id: TermId, constraint_type: RefinedBase) -> (Self,
549     -> Constraint) {
550     let term = &self.body.expressions[term_id];
551     match term {
552         Term::Block {
553             statements,
554             trailing_expression,
555         } => {

```

```

555     let (fold, constraints) =
556         statements
557         .iter()
558         .fold(
559             (self, vec![]),
560             |(fold, mut constraints), statement| match statement {
561                 Statement::Let(pattern, init_term) => {
562                     let (mut fold, ty, c) = fold.synth_type(*init_term);
563                     let pattern = &fold.body.patterns[*pattern];
564                     let (path, name) = match pattern {
565                         Pattern::Bind(name) => (
566                             Path {
567                                 segments: vec![name.clone()],
568                             },
569                             name.clone(),
570                         ),
571                         _ => todo!(),
572                     };
573                     fold.context.bindings.push((path, ty.clone()));
574                     constraints.push((name, ty, c));
575                     (fold, constraints)
576                 }
577                 Statement::Expression(_) => todo!(),
578             },
579         );
580         let (fold, c) = fold.check_type(*trailing_expression, constraint_type);
581         let constraint = constraints.into_iter().rfold(
582             Some(c),
583             |inner_constraint, (name, ty, outer_constraint)| {
584                 let implication_constraint =
585                     implication_constraint(name, ty.as_refined_base(),
586                     ↣ inner_constraint);
587                 Some(Constraint::make_conjunction(
588                     outer_constraint,
589                     implication_constraint,
590                 ))
591             },
592             (fold, constraint.unwrap())
593         }
594         Term::If {
595             condition,
596             then_branch,
597             else_branch,
598         } => {
599             let fresh_var = self.make_fresh_var();
600             let (fold, constraint) = self.check_type(
601                 *condition,

```

```

602             RefinedBase {
603                 base: Type::Scalar(ScalarType::Boolean),
604                 binder: Name::new_inline("b"),
605                 predicate: Predicate::Boolean(true),
606             },
607         );
608         if !entailment(fold.context.clone(), constraint) {
609             panic!("if condition not a valid boolean")
610         }
611
612         let (fold, constraint) = fold.check_type(*then_branch,
613             ↪ constraint_type.clone());
613         let c1 = implication_constraint(
614             fresh_var.clone(),
615             RefinedBase {
616                 base: Type::Scalar(ScalarType::Boolean),
617                 binder: Name::new_inline("branch_"),
618                 predicate: Predicate::Variable(fold.term_as_name(*condition)),
619             },
620             Some(constraint),
621         );
622         let (fold, constraint) = fold.check_type(*else_branch, constraint_type);
623         let c2 = implication_constraint(
624             fresh_var,
625             RefinedBase {
626                 base: Type::Scalar(ScalarType::Boolean),
627                 binder: Name::new_inline("branch_"),
628                 predicate: Predicate::Unary(
629                     UnaryOperator::Negation,
630                     Predicate::Variable(fold.term_as_name(*condition)).into(),
631                 ),
632             },
633             Some(constraint),
634         );
635         (fold, Constraint::Conjunction(c1.into(), c2.into()))
636     }
637     - => {
638         let (fold, t, c) = self.synth_type(term_id);
639         let (fold, c2) = fold.subtype(t.as_refined_base(), constraint_type);
640         (fold, Constraint::make_conjunction(c, c2))
641     }
642 }
643 }
644
645 fn synth_type(self, term_id: TermId) -> (Self, RefinedType, Option<Constraint>) {
646     let term = &self.body.expressions[term_id];
647     match term {
648         Term::Block {

```

```

649         statements,
650         trailing_expression,
651     } => {
652         let (fold, constraints) =
653             statements
654                 .iter()
655                 .fold(
656                     (self, vec![]),
657                     |(fold, mut constraints), statement| match statement {
658                         Statement::Let(pattern, init_term) => {
659                             let (mut fold, ty, c) = fold.synth_type(*init_term);
660                             let pattern = &fold.body.patterns[*pattern];
661                             let (path, name) = match pattern {
662                                 Pattern::Bind(name) => (
663                                     Path {
664                                         segments: vec![name.clone()],
665                                         },
666                                         name.clone(),
667                                         ),
668                                         _ => todo!(),
669                                     );
670                             fold.context.bindings.push((path, ty.clone()));
671                             constraints.push((name, ty, c));
672                             (fold, constraints)
673                         }
674                         Statement::Expression(_) => todo!(),
675                     },
676                 );
677         let (fold, base, c) = fold.synth_type(*trailing_expression);
678         let constraint = constraints.into_iter().rfold(
679             c,
680             |inner_constraint, (name, ty, outer_constraint)| {
681                 let implication_constraint =
682                     implication_constraint(name, ty.as_refined_base(),
683                     ↣ inner_constraint);
684                 Some(Constraint::make_conjunction(
685                     outer_constraint,
686                     implication_constraint,
687                     ))
688                 },
689                 (fold, base, constraint)
690             }
691         Term::Path(path) => match self
692             .context
693             .get(path)
694             .expect(format!("{} not in context.").as_str())
695             .clone()

```

```

696     {
697         RefinedType::Base(RefinedBase {
698             base,
699             binder,
700             predicate,
701         }) => {
702             assert!(binder != path.as_name());
703
704             let path_type = RefinedBase {
705                 base,
706                 binder: binder.clone(),
707                 predicate: Predicate::Binary(
708                     BinaryOperator::Logic(LogicOperator::And),
709                     predicate.into(),
710                     Predicate::Binary(
711                         BinaryOperator::Compare(CompareOperator::Equality {
712                             negated: false,
713                         }),
714                         Predicate::Variable(binder).into(),
715                         Predicate::Variable(path.as_name()).into(),
716                         )
717                         .into(),
718                     ),
719                 );
720                 (self, path_type.into(), None)
721             }
722             dependent_fn => (self, dependent_fn, None),
723         },
724         Term::Literal(lit) => match lit {
725             Literal::Integer(value, sufix) => {
726                 (self, primitive_integer(*value, *sufix).into(), None)
727             }
728             Literal::Bool(value) => (self, primitive_bool(*value).into(), None),
729         },
730         Term::Unary(op, term) => self.synth_unary_term(op, *term),
731         Term::Binary(op, lhs, rhs) => self.synth_binary_term(op, *lhs, *rhs).into(),
732         Term::Call { callee, arguments } => {
733             self.synth_call_term(*callee, arguments.clone()).into()
734         }
735         term => panic!("Unhandled term {:?}", term),
736     }
737 }
738
739 pub(crate) fn synth_unary_term(
740     self,
741     op: &'a TermUnaryOp,
742     term: IIdx<Term>,
743 ) -> (Fold, RefinedType, Option<Constraint>) {

```

```

744 match op {
745   TermUnaryOp::Minus => {
746     let minus_signature = DependentFunction {
747       parameter: (
748         Name::new_inline("param0"),
749         RefinedBase {
750           base: Type::Scalar(ScalarType::Integer(IntegerKind::I64)),
751           binder: Name::new_inline("param0"),
752           predicate: Predicate::Boolean(true),
753         },
754       ),
755       tail_type: RefinedType::Base(RefinedBase {
756         base: Type::Scalar(ScalarType::Integer(IntegerKind::I64)),
757         binder: Name::new_inline("ret"),
758         predicate: Predicate::Binary(
759           BinaryOperator::Compare(CompareOperator::Equality {
760             ↪ negated: false }),
761             Predicate::Variable(Name::new_inline("ret")).into(),
762             Predicate::Unary(
763               UnaryOperator::Minus,
764               Predicate::Variable(Name::new_inline("param0")).into(),
765             )
766             .into(),
767           ),
768           .into(),
769         );
770
771       let (fold, ty, constraint) =
772         self.synth_function_call(minus_signature, vec![term], None);
773       (fold, ty, constraint)
774     }
775   TermUnaryOp::Negation => {
776     let minus_signature = DependentFunction {
777       parameter: (
778         Name::new_inline("param0"),
779         RefinedBase {
780           base: Type::Scalar(ScalarType::Boolean),
781           binder: Name::new_inline("param0"),
782           predicate: Predicate::Boolean(true),
783         },
784       ),
785       tail_type: RefinedType::Base(RefinedBase {
786         base: Type::Scalar(ScalarType::Boolean),
787         binder: Name::new_inline("ret"),
788         predicate: Predicate::Binary(
789           BinaryOperator::Compare(CompareOperator::Equality {
790             ↪ negated: false }),
```

```

790             Predicate::Variable(Name::new_inline("ret")).into(),
791             Predicate::Unary(
792                 UnaryOperator::Negation,
793                 Predicate::Variable(Name::new_inline("param0")).into(),
794             )
795             .into(),
796         ),
797     )
798     .into(),
799 };
800
801     let (fold, ty, constraint) =
802         self.synth_function_call(minus_signature, vec![term], None);
803     (fold, ty, constraint)
804 }
805     TermUnaryOp::Reference => todo!(),
806     TermUnaryOp::Dereference => todo!(),
807 }
808 }
809
810 fn synth_binary_term(
811     self,
812     op: &'a BinaryOperator,
813     lhs: Idx<Term>,
814     rhs: Idx<Term>,
815 ) -> (Fold, RefinedType, Option<Constraint>) {
816     let lhs_ty = self.inference.type_of_expression[lhs].clone();
817     let rhs_ty = self.inference.type_of_expression[rhs].clone();
818     let sum_signature = DependentFunction {
819         parameter: (
820             Name::new_inline("param0"),
821             RefinedBase {
822                 base: lhs_ty.clone(),
823                 binder: Name::new_inline("param0"),
824                 predicate: Predicate::Boolean(true),
825             },
826         ),
827         tail_type: RefinedType::Fn(DependentFunction {
828             parameter: (
829                 Name::new_inline("param1"),
830                 RefinedBase {
831                     base: rhs_ty,
832                     binder: Name::new_inline("param1"),
833                     predicate: Predicate::Boolean(true),
834                 },
835             ),
836             tail_type: RefinedType::Base(RefinedBase {
837                 base: match op {

```

```

838             BinaryOperator::Arithmetic(_) => lhs_ty,
839             BinaryOperator::Logic(_) | BinaryOperator::Compare(_) => {
840                 Type::Scalar(ScalarType::Boolean)
841             }
842         },
843         binder: Name::new_inline("ret"),
844         predicate: Predicate::Binary(
845             BinaryOperator::Compare(CompareOperator::Equality { negated:
846                 false })),
847             Predicate::Variable(Name::new_inline("ret")).into(),
848             Predicate::Binary(
849                 *op,
850                 Predicate::Variable(Name::new_inline("param0")).into(),
851                 Predicate::Variable(Name::new_inline("param1")).into(),
852             )
853             .into(),
854         ),
855         .into(),
856     );
857
858     let (fold, ty, constraint) = self.synth_function_call(sum_signature, vec![lhs,
859         rhs], None);
860     (fold, ty, constraint)
861 }
862
863 fn synth_call_term(
864     self,
865     callee: IIdx<Term>,
866     arguments: Vec<IIdx<Term>>,
867 ) -> (Self, RefinedType, Option<Constraint>) {
868     let (fold, function_ty, constraint) = self.synth_type(callee);
869
870     let (fold, ret_ty, constraint) =
871         fold.synth_function_call(function_ty.as_dependent_function(), arguments,
872             constraint);
873     (fold, ret_ty, constraint)
874 }
875
876 fn synth_function_call(
877     self,
878     function_ty: DependentFunction,
879     mut argument_terms: Vec<TermId>,
880     constraint: Option<Constraint>,
881 ) -> (Self, RefinedType, Option<Constraint>) {
882     if argument_terms.is_empty() {

```

```

883         (self, function_ty.into(), constraint)
884     } else {
885         let argument = argument_terms.pop().unwrap();
886         // synth
887         let (fold, ty, constraint) =
888             self.synth_function_call(function_ty, argument_terms, constraint);
889
890         let function_ty = ty.as_dependent_function();
891         let (parameter_name, argument_type) = function_ty.parameter;
892
893         // check
894         let (fold, c) = fold.check_type(argument, argument_type.clone());
895
896         let argument_name = fold.term_as_name(argument);
897
898         // substitute
899         let return_type =
900             substitution_in_refined_type(*function_ty.tail_type, parameter_name,
901                                         ↳ argument_name);
902         (
903             fold,
904             return_type.into(),
905             Some(Constraint::make_conjunction(constraint, c)),
906         )
907     }
908
909     fn make_fresh_var(&mut self) -> Name {
910         let fresh = self.fresh_var_counter;
911         self.fresh_var_counter += 1;
912         Name {
913             id: format!("__fresh{}", fresh).into(),
914         }
915     }
916
917     fn term_as_name(&self, term_id: TermId) -> Name {
918         match &self.body.expressions[term_id] {
919             Term::Path(path) => path.as_name(),
920             _ => panic!("not a path term"),
921         }
922     }
923 }
924
925 fn substitution_in_refined_base(ty: RefinedBase, old_name: Name, new_name: Name) ->
926     ↳ RefinedBase {
927     let RefinedBase {
928         base,
929         binder,

```

```

929         predicate,
930     } = ty;
931     if new_name == binder {
932         let new_binder = Name::new_inline(format!("{}{}", binder.id.as_str(),
933                                         & 1).as_str());
934         substitution_in_refined_base(
935             RefinedBase {
936                 base,
937                 binder: new_binder.clone(),
938                 predicate: substitution(binder, new_binder, predicate),
939             },
940             old_name,
941             new_name,
942         )
943     } else if old_name == binder {
944         RefinedBase {
945             base,
946             binder,
947             predicate,
948         }
949     } else {
950         RefinedBase {
951             base,
952             binder,
953             predicate: substitution(old_name, new_name, predicate),
954         }
955     }
956
957 fn substitution_in_function_type(
958     function_ty: DependentFunction,
959     old_name: Name,
960     new_name: Name,
961 ) -> DependentFunction {
962     let DependentFunction {
963         parameter: (arg_name, arg_type),
964         tail_type,
965     } = function_ty;
966     if new_name == arg_name {
967         let new_arg_name = Name::new_inline(format!("{}{}", arg_name.id.as_str(),
968                                         & 1).as_str());
969         substitution_in_function_type(
970             DependentFunction {
971                 parameter: (
972                     new_arg_name.clone(),
973                     substitution_in_refined_base(arg_type, arg_name, new_arg_name),
974                 ),
975                 tail_type,

```

```

975         },
976         old_name,
977         new_name,
978     )
979 } else if old_name == arg_name {
980     DependentFunction {
981         parameter: (
982             arg_name,
983             substitution_in_refined_base(arg_type, old_name, new_name),
984         ),
985         tail_type,
986     }
987 } else {
988     DependentFunction {
989         parameter: (
990             arg_name,
991             substitution_in_refined_base(arg_type, old_name.clone(),
992                 new_name.clone()),
993             tail_type: substitution_in_refined_type(*tail_type, old_name,
994                 new_name).into(),
995         )
996     }
997 }
998 fn substitution_in_refined_type(ty: RefinedType, old_name: Name, new_name: Name) ->
999     RefinedType {
1000     match ty {
1001         RefinedType::Base(base) => substitution_in_refined_base(base, old_name,
1002             new_name).into(),
1003         RefinedType::Fn(func) => substitution_in_function_type(func, old_name,
1004             new_name).into(),
1005     }
1006 }
1007 fn primitive_bool(value: bool) -> RefinedBase {
1008     let binder = Name::new_inline("lit");
1009     RefinedBase {
1010         base: Type::Scalar(ScalarType::Boolean),
1011         binder: binder.clone(),
1012         predicate: match value {
1013             true => Predicate::Variable(binder),
1014             false => Predicate::Unary(UnaryOperator::Negation,
1015                 Predicate::Variable(binder).into()),
1016         },
1017     }
1018 }
```

```

1017 fn primitive_integer(value: u128, sufix: Option<BuiltInInteger>) -> RefinedBase {
1018     let binder = Name::new_inline("lit");
1019     RefinedBase {
1020         base: Type::Scalar(ScalarType::Integer(
1021             sufix.map_or(IntegerKind::I64, Into::into),
1022         )),
1023         binder: binder.clone(),
1024         predicate: Predicate::Binary(
1025             BinaryOperator::Compare(CompareOperator::Equality { negated: false }),
1026             Predicate::Variable(binder).into(),
1027             Predicate::Integer(value).into(),
1028         ),
1029     }
1030 }
1031
1032 fn substitution(old: Name, new: Name, predicate: Predicate) -> Predicate {
1033     match predicate.clone() {
1034         Predicate::Variable(name) => {
1035             if name == old {
1036                 Predicate::Variable(new)
1037             } else {
1038                 predicate
1039             }
1040         }
1041         Predicate::Binary(op, lhs, rhs) => Predicate::Binary(
1042             op,
1043             substitution(old.clone(), new.clone(), *lhs).into(),
1044             substitution(old, new, *rhs).into(),
1045         ),
1046         Predicate::Unary(op, predicate) => {
1047             Predicate::Unary(op, substitution(old, new, *predicate).into())
1048         }
1049         Predicate::Boolean(_) | Predicate::Integer(_) => predicate,
1050     }
1051 }
1052
1053 fn implication_constraint(
1054     name: Name,
1055     ty: RefinedBase,
1056     constraint: Option<Constraint>,
1057 ) -> Constraint {
1058     let RefinedBase {
1059         base,
1060         binder,
1061         predicate,
1062     } = ty;
1063
1064     let constraint = Constraint::Implication {

```

```

1065     binder: name.clone(),
1066     base,
1067     antecedent: substitution(binder, name, predicate),
1068     consequent: constraint
1069         .unwrap_or(Constraint::Predicate(Predicate::Boolean(true)))
1070         .into(),
1071     );
1072
1073     constraint
1074 }
```

A.2 MODULE CODEGEN

```

1 use std::{
2     cell::RefCell,
3     collections::{BTreeMap, HashMap},
4     fmt::Display,
5     iter::FromIterator,
6     sync::Arc,
7 };
8
9 use by_address::ByAddress;
10 use inkwell::{
11     attributes::{Attribute, AttributeLoc},
12     basic_block::BasicBlock,
13     builder::Builder,
14     context::Context,
15     module::Module,
16     targets::{
17         CodeModel, InitializationConfig, RelocMode, Target, TargetData, TargetMachine,
18         → TargetTriple,
19     },
20     types::{AnyType, BasicType, BasicTypeEnum, FunctionType, IntType, StructType},
21     values::{BasicValue, BasicValueEnum, FunctionValue, PointerValue},
22     AddressSpace, IntPredicate, OptimizationLevel,
23 };
24 use hir::{
25     check::type_inference::InferenceResult,
26     check::{IntegerKind, ScalarType, Type},
27     semantic_ir::{
28         definition_map::{CallableDefinitionId, ValueConstructorId},
29         name::Name,
30         path_resolver::{Resolver, ValueNamespaceItem},
31     },
32     semantic_ir::{
33         definition_map::{FunctionDefinitionId, TypeDefinitionId},

```

```

34     path::Path,
35     term::{
36         ArithmeticOperator, BinaryOperator, Body, CompareOperator, Literal,
37             ↪ LogicOperator,
38         Ordering, Pattern, PatternId, Statement, Term, TermId, UnaryOperator,
39     },
40     HirDatabase, SourceDatabase, Upcast,
41 };
42
43 #[salsa::query_group(CodeGenDatabaseStorage)]
44 pub trait CodeGenDatabase: HirDatabase + Upcast<dyn HirDatabase> {
45     #[salsa::input]
46     fn target(&self) -> CodeGenTarget;
47
48     #[salsa::input]
49     fn liquid(&self) -> bool;
50
51     fn target_machine(&self) -> ByAddress<Arc<TargetMachine>>;
52
53     fn build_assembly_ir(&self) -> ();
54 }
55
56 fn target_machine(db: &dyn CodeGenDatabase) -> ByAddress<Arc<TargetMachine>> {
57     let target = db.target();
58
59     match target.triple.arch {
60         CodeGenTargetArch::X86_64 => {
61             Target::initialize_x86(&InitializationConfig::default());
62         }
63     };
64
65     let target_triple = TargetTriple::create(&target.triple.to_string());
66     let target = Target::from_triple(&target_triple).unwrap_or_else(|err| {
67         panic!(
68             "Could not create LLVM target from target triple {}: {}",
69             target.triple
70         )
71     });
72     let opt = OptimizationLevel::None;
73     let reloc = RelocMode::Default;
74     let model = CodeModel::Default;
75     let cpu = "";
76     let features = "";
77     let target_machine = target
78         .create_target_machine(&target_triple, cpu, features, opt, reloc, model)
79         .unwrap();
80 }
```

```

81     ByAddress(Arc::new(target_machine))
82 }
83
84 #[salsa::database(
85     hir::SourceDatabaseStorage,
86     hir::InternerStorage,
87     hir::DefinitionsDatabaseStorage,
88     hir::HirDatabaseStorage,
89     CodeGenDatabaseStorage
90 )]
91 #[derive(Default)]
92 pub struct Database {
93     storage: salsa::Storage<Self>,
94 }
95
96 impl Upcast<dyn hir::SourceDatabase> for Database {
97     fn upcast(&self) -> &(dyn hir::SourceDatabase + 'static) {
98         &*self
99     }
100 }
101
102 impl Upcast<dyn hir::Interner> for Database {
103     fn upcast(&self) -> &(dyn hir::Interner + 'static) {
104         &*self
105     }
106 }
107
108 impl Upcast<dyn hir::DefinitionsDatabase> for Database {
109     fn upcast(&self) -> &(dyn hir::DefinitionsDatabase + 'static) {
110         &*self
111     }
112 }
113
114 impl Upcast<dyn hir::HirDatabase> for Database {
115     fn upcast(&self) -> &(dyn hir::HirDatabase + 'static) {
116         &*self
117     }
118 }
119
120 impl salsa::Database for Database {}
121
122 #[derive(Clone)]
123 struct LocalBindingStack<'ink> {
124     stack: Vec<Scope<'ink>>,
125 }
126
127 impl<'ink> LocalBindingStack<'ink> {
128     pub fn get(&self, name: &Name) -> Option<Value<'ink>> {

```

```

129         self.stack.iter().rev().find_map(|scope| scope.get(name))
130     }
131
132     pub fn push(&mut self, scope: Scope<'ink>) {
133         self.stack.push(scope);
134     }
135
136     pub fn pop(&mut self) {
137         self.stack.pop();
138     }
139 }
140
141 impl<'ink> FromIterator<Scope<'ink>> for LocalBindingStack<'ink> {
142     fn from_iter<I>(iter: I) -> Self
143     where
144         I: IntoIterator<Item = Scope<'ink>>,
145     {
146         let stack = iter.into_iter().collect();
147         Self { stack }
148     }
149 }
150
151 #[derive(Clone)]
152 struct Scope<'ink> {
153     scope: Vec<Binding<'ink>>,
154 }
155
156 impl<'ink> Scope<'ink> {
157     pub fn get(&self, name: &Name) -> Option<Value<'ink>> {
158         self.scope
159             .iter()
160             .find_map(|binding| match &binding.name == name {
161                 true => Some(binding.value.clone()),
162                 false => None,
163             })
164     }
165 }
166
167 impl<'ink> FromIterator<Binding<'ink>> for Scope<'ink> {
168     fn from_iter<I>(iter: I) -> Self
169     where
170         I: IntoIterator<Item = Binding<'ink>>,
171     {
172         let scope = iter.into_iter().collect();
173         Self { scope }
174     }
175 }
176

```

```

177 #[derive(Clone)]
178 struct Binding<'ink> {
179     pub name: Name,
180     pub value: Value<'ink>,
181 }
182
183 #[derive(Debug, Clone)]
184 struct Value<'a> {
185     pub kind: ValueKind,
186     pub value: BasicValueEnum<'a>,
187 }
188
189 impl<'a> Value<'a> {
190     fn new(kind: ValueKind, value: BasicValueEnum<'a>) -> Self {
191         Self { kind, value }
192     }
193 }
194
195 #[derive(Debug, Clone, Copy)]
196 pub enum CodeGenTargetArch {
197     X86_64,
198 }
199
200 impl Display for CodeGenTargetArch {
201     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
202         let string = match self {
203             Self::X86_64 => "x86_64",
204         };
205         f.write_str(string)
206     }
207 }
208
209 #[derive(Debug, Clone, Copy)]
210 pub enum CodeGenTargetVendor {
211     PC,
212 }
213
214 impl Display for CodeGenTargetVendor {
215     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
216         let string = match self {
217             Self::PC => "pc",
218         };
219         f.write_str(string)
220     }
221 }
222
223 #[derive(Debug, Clone, Copy)]
224 pub enum CodeGenTargetSystem {

```

```

225     Linux,
226 }
227
228 impl Display for CodeGenTargetSystem {
229     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
230         let string = match self {
231             Self::Linux => "linux",
232         };
233         f.write_str(string)
234     }
235 }
236
237 #[derive(Debug, Clone, Copy)]
238 pub enum CodeGenTargetABI {
239     GNU,
240 }
241
242 impl Display for CodeGenTargetABI {
243     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
244         let string = match self {
245             Self::GNU => "gnu",
246         };
247         f.write_str(string)
248     }
249 }
250
251 #[derive(Debug, Clone, Copy)]
252 pub struct CodeGenTargetTriple {
253     pub arch: CodeGenTargetArch,
254     pub vendor: CodeGenTargetVendor,
255     pub system: CodeGenTargetSystem,
256     pub abi: CodeGenTargetABI,
257 }
258
259 impl Display for CodeGenTargetTriple {
260     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
261         f.write_fmt(format_args!(
262             "{}-{}-{}-{}",
263             self.arch, self.vendor, self.system, self.abi
264         ))
265     }
266 }
267
268 #[derive(Debug, Clone, Copy)]
269 pub struct CodeGenTarget {
270     pub triple: CodeGenTargetTriple,
271 }
272

```

```

273 pub struct CodeGenTypeCache<'db, 'context> {
274     db: &'db dyn CodeGenDatabase,
275     context: &'context Context,
276     target_data: TargetData,
277     adt_map: RefCell<HashMap<TypeID, (StructType<'context>,
278                                TypeInfo<'context>)>>,
279     adt_variant_map: RefCell<HashMap<ValueConstructorID, StructType<'context>>>,
280 }
281
282 impl<'db, 'context> CodeGenTypeCache<'db, 'context> {
283     pub(crate) fn new(db: &'db dyn CodeGenDatabase, context: &'context Context) -> Self {
284         {
285             Self {
286                 db,
287                 context,
288                 target_data: db.target_machine().get_target_data(),
289                 adt_map: RefCell::new(HashMap::new()),
290                 adt_variant_map: RefCell::new(HashMap::new()),
291             }
292         }
293
294         fn llvm_type(&self, ty: &Type) -> BasicTypeEnum<'context> {
295             match ty {
296                 Type::AbstractDataType(ty_loc_id) =>
297                     self.adt_struct_type(*ty_loc_id).into(),
298                 Type::FunctionDefinition(_) => todo!(),
299                 Type::Scalar(scalar) => self.scalar_type(scalar),
300                 Type::Pointer(inner) => {
301                     let inner = self.llvm_type(inner);
302                     let ptr_type = inner.ptr_type(AddressSpace::Generic);
303                     ptr_type.as_basic_type_enum()
304                 }
305             }
306         }
307
308         fn type_info(&self, ty: &Type) -> TypeInfo<'context> {
309             match ty {
310                 Type::AbstractDataType(ty_loc_id) => self.adt_struct_info(*ty_loc_id),
311                 Type::FunctionDefinition(_) => todo!(),
312                 Type::Scalar(_) => todo!(),
313                 Type::Pointer(_) => todo!(),
314             }
315
316         fn bit_size(&self, ty: &Type) -> usize {
317             let llvm_type = self.llvm_type(ty);
318             self.target_data.get_bit_size(&llvm_type.as_any_type_enum()) as usize
319         }

```

```

318
319     fn target_data(&self) -> &TargetData {
320         &self.target_data
321     }
322
323     fn adt_struct_info(&self, ty_loc_id: TypeDefinitionId) -> TypeInfo<'context> {
324         self.adt_struct(ty_loc_id).1
325     }
326
327     fn adt_struct_type(&self, ty_loc_id: TypeDefinitionId) -> StructType<'context> {
328         self.adt_struct(ty_loc_id).0
329     }
330
331     fn adt_struct(
332         &self,
333         ty_loc_id: TypeDefinitionId,
334     ) -> (StructType<'context>, TypeInfo<'context>) {
335         if let Some(value) = self.adt_map.borrow().get(&ty_loc_id) {
336             return value.clone();
337         };
338         let ty_data = self.db.type_definition_data(ty_loc_id);
339         let opaque = self.context.opaque_struct_type(ty_data.name.id.as_str());
340         self.adt_map
341             .borrow_mut()
342             .insert(ty_loc_id, (opaque, TypeInfo::default()));
343
344         let tag_bit_len = {
345             let variant_count = ty_data.value_constructors.len();
346             let mut bit_len = 0;
347             while 1 << bit_len < variant_count {
348                 bit_len += 1;
349             }
350             bit_len
351         };
352
353         let tag_type = match tag_bit_len {
354             0 => Some(self.context.i64_type()),
355             1 => Some(self.context.i64_type()),
356             2..=8 => Some(self.context.i64_type()),
357             9..=16 => Some(self.context.i64_type()),
358             17..=32 => Some(self.context.i64_type()),
359             33..=64 => Some(self.context.i64_type()),
360             _ => panic!("Tag for ADT too big! {tag_bit_len:?}"),
361         };
362
363         let value_constructor_ids = ty_data
364             .value_constructors
365             .iter()

```

```

366         .map(|(id, _)| ValueConstructorId {
367             type_definition_id: ty_loc_id,
368             id,
369         })
370         .collect::<Vec<_>>();
371
372     let ty_constructors = value_constructor_ids
373         .iter()
374         .cloned()
375         .map(|id| self.value_constructor_struct(id));
376
377     let bigest = ty_constructors
378         .max_by_key(|struct_type| {
379             self.target_data
380                 .get_abi_size(&struct_type.as_any_type_enum())
381         })
382         .expect("Failed to get memsize of type variant");
383
384     let field_types = tag_type
385         .into_iter()
386         .map(Into::into)
387         .chain(std::iter::once(bigest.into()))
388         .collect::<Vec<_>>();
389
390     opaque.set_body(field_types.as_slice(), false);
391     let tag_map = value_constructor_ids
392         .into_iter()
393         .map(|id| (id, u32::from(id.id.into_raw()) as usize))
394         .collect();
395     let adt_info = TypeInfo {
396         tag: tag_type,
397         tag_map,
398     };
399     let value = (opaque, adt_info);
400     self.adt_map.borrow_mut().insert(ty_loc_id, value.clone());
401     value
402 }
403
404 fn value_constructor_struct(&self, constructor_id: ValueConstructorId) ->
405     StructType<'context> {
406     if let Some(struct_type) = self.adt_variant_map.borrow().get(&constructor_id) {
407         return *struct_type;
408     }
409
410     let constructor = self.db.callable_definition_signature(constructor_id.into());
411     let struct_data = constructor
412         .parameter_types
413         .iter()

```

```

413         .map(|ty| self.llvm_type(ty))
414         .collect::<Vec<_>>();
415
416     let ty_data = self
417         .db
418         .type_definition_data(constructor_id.type_definition_id);
419     let variant_data = ty_data.value_constructor(constructor_id.id);
420
421     let opaque = self
422         .context
423         .opaque_struct_type(format!("{}:{}", ty_data.name.id,
424                                     variant_data.name.id).as_str());
425
426     opaque.set_body(&struct_data, false);
427
428     self.adt_variant_map
429         .borrow_mut()
430         .insert(constructor_id, opaque);
431     opaque
432 }
433
434 fn scalar_type(&self, scalar: &ScalarType) -> BasicTypeEnum<'context> {
435     match scalar {
436         ScalarType::Integer(int_kind) => match int_kind {
437             IntegerKind::I32 => self.context.i32_type().into(),
438             IntegerKind::I64 => self.context.i64_type().into(),
439         },
440         ScalarType::Boolean => self.context.bool_type().into(),
441     }
442 }
443
444 #[derive(Default, Clone)]
445 struct TypeInfo<'context> {
446     pub tag: Option<IntType<'context>>,
447     pub tag_map: HashMap<ValueConstructorId, usize>,
448 }
449
450 struct CodeGenFunctionInfoCache<'db, 'context, 'type_cache> {
451     db: &'db dyn CodeGenDatabase,
452     context: &'context Context,
453     type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
454 }
455
456 impl<'db, 'context, 'type_cache> CodeGenFunctionInfoCache<'db, 'context, 'type_cache>
457     {
458         pub(crate) fn new(
459             db: &'db dyn CodeGenDatabase,

```

```
459     context: &'context Context,
460     type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
461 ) -> Self {
462     Self {
463         db,
464         context,
465         type_cache,
466     }
467 }
468
469 pub(crate) fn llvm_function_type(
470     &self,
471     function_id: &FunctionDefinitionId,
472 ) -> FunctionType<'context> {
473     let function_info = self.function_info(function_id);
474     let function_signature =
475         ↳ self.db.callable_definition_signature((*function_id).into());
476     let parameter_types = function_signature
477         .parameter_types
478         .iter()
479         .zip(function_info.parameter_kinds)
480         .map(|(param_type, param_kind)| {
481             let llvm_param_type = self.type_cache.llvm_type(param_type);
482             match param_kind {
483                 ValueKind::Indirect => llvm_param_type
484                     .ptr_type(AddressSpace::Generic)
485                     .as_basic_type_enum(),
486                 ValueKind::Direct => llvm_param_type,
487             }
488         });
489     let return_type = {
490         let return_type = function_signature.return_type;
491         self.type_cache.llvm_type(&return_type)
492     };
493     match function_info.return_kind {
494         ValueKind::Indirect => {
495             let return_type = return_type
496                 .ptr_type(AddressSpace::Generic)
497                 .as_basic_type_enum();
498             let parameter_types =
499                 ↳ std::iter::once(return_type).chain(parameter_types);
500             self.context void_type().fn_type(
501                 parameter_types
502                     .map(Into::into)
503                     .collect::<Vec<_>>()
504                     .as_slice(),
505                     false,
506             )
507         }
508     };
509 }
```

```

505     }
506     ValueKind::Direct => return_type.fn_type(
507         &parameter_types
508             .into_iter()
509             .map(Into::into)
510             .collect::<Vec<_>>(),
511             false,
512         ),
513     }
514 }
515
516 pub(crate) fn function_info(&self, function_id: &FunctionDefinitionId) ->
517     FunctionInfo {
518     let free_integer_registers = 6;
519     let function_signature =
520         self.db.callable_definition_signature(function_id);
521     let return_bitsize = self.type_cache.bit_size(&function_signature.return_type);
522     let (free_integer_registers, return_kind) = match return_bitsize {
523         0..=128 => (free_integer_registers, ValueKind::Direct),
524         _ => (free_integer_registers - 1, ValueKind::Indirect),
525     };
526
527     let (_ , parameter_kinds) = function_signature.parameter_types.iter().fold(
528         (free_integer_registers, Vec::new()),
529         |(free_integer_registers, mut param_kinds), param_type| match
530             free_integer_registers {
531                 0 => {
532                     let bitsize = self.type_cache.bit_size(param_type);
533                     if bitsize <= 64 {
534                         param_kinds.push(ValueKind::Direct)
535                     } else {
536                         param_kinds.push(ValueKind::Indirect);
537                     }
538                     (free_integer_registers, param_kinds)
539                 }
540             } => {
541                 let param_size = self.type_cache.bit_size(param_type);
542                 let (free_integer_registers, param_kind) = match param_size {
543                     1..=64 => (free_integer_registers - 1, ValueKind::Direct),
544                     65..=128 => (free_integer_registers - 2, ValueKind::Direct),
545                     129.. => (free_integer_registers, ValueKind::Indirect),
546                     x => panic!("parameter size {x} not supported"),
547                 };
548                 param_kinds.push(param_kind);
549                 (free_integer_registers, param_kinds)
550             }
551         },
552     );
553 }

```

```

550     FunctionInfo {
551         return_kind,
552         parameter_kinds,
553     }
554 }
555 }
556
557 struct FunctionInfo {
558     pub return_kind: ValueKind,
559     pub parameter_kinds: Vec<ValueKind>,
560 }
561
562 impl FunctionInfo {
563     pub(crate) fn skip_return_param<T>(
564         &self,
565         iter: impl Iterator<Item = T>,
566     ) -> impl Iterator<Item = T> {
567         iter.skip(match self.return_kind {
568             ValueKind::Indirect => 1,
569             ValueKind::Direct => 0,
570         })
571     }
572
573     pub(crate) fn get_return_type<'ctx>(
574         &self,
575         function_value: FunctionValue<'ctx>,
576     ) -> BasicTypeEnum<'ctx> {
577         match self.return_kind {
578             ValueKind::Indirect => function_value.get_type().get_param_types()[0],
579             ValueKind::Direct => function_value.get_type().get_return_type().unwrap(),
580         }
581     }
582 }
583
584 struct CodeGenFunctionValueCache<'db, 'context, 'module, 'type_cache,
585     → 'function_info_cache> {
586     db: &'db dyn CodeGenDatabase,
587     context: &'context Context,
588     module: &'module Module<'context>,
589     type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
590     function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<'db, 'context,
591     → 'type_cache>,
592     function_value_map: RefCell<HashMap<FunctionDefinitionId,
593     → FunctionValue<'context>>>,
594 }
595
596 impl<'db, 'context, 'module, 'type_cache, 'function_info_cache>

```

```

594     CodeGenFunctionValueCache<'db, 'context, 'module, 'type_cache,
595     ↪  'function_info_cache>
596 {
597     pub(crate) fn new(
598         db: &'db dyn CodeGenDatabase,
599         context: &'context Context,
600         module: &'module Module<'context>,
601         type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
602         function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<
603             'db,
604             'context,
605             'type_cache,
606         >,
607     ) -> Self {
608         Self {
609             db,
610             context,
611             module,
612             type_cache,
613             function_info_cache,
614             function_value_map: RefCell::new(HashMap::new()),
615         }
616     }
617
618     fn llvm_function_value(&self, function_id: &FunctionDefinitionId) ->
619         FunctionValue<'context> {
620         if let Some(function_value) =
621             self.function_value_map.borrow().get(function_id) {
622             return *function_value;
623         }
624
625         let name = {
626             let f_data = self.db.function_definition_data(*function_id);
627             f_data.name.id
628         };
629         let llvm_function_type =
630             self.function_info_cache.llvm_function_type(function_id);
631         let function_value = self
632             .module
633             .add_function(name.as_str(), llvm_function_type, None);
634
635         let function_info = self.function_info_cache.function_info(function_id);
636
637         match function_info.return_kind {
638             ValueKind::Indirect => {
639                 let ret_ty = llvm_function_type
640                     .get_param_types()
641                     .first()
642             }
643         }
644     }
645 }

```

```

638         .unwrap()
639         .into_pointer_type()
640         .get_element_type();
641     let kind_id = Attribute::get_named_enum_kind_id("sret");
642     let sret_attribute = self
643         .context
644         .create_type_attribute(kind_id, ret_ty.as_any_type_enum());
645     let kind_id = Attribute::get_named_enum_kind_id("noalias");
646     let noalias_attribute = self.context.create_enum_attribute(kind_id, 0);
647     function_value.add_attribute(AttributeLoc::Param(0), sret_attribute);
648     function_value.add_attribute(AttributeLoc::Param(0), noalias_attribute);
649 }
650 _ => (),
651 };
652
653 for ((param_index, param_type), param_kind) in function_info
654     .skip_return_param(llvm_function_type.get_param_types().iter().enumerate())
655     .zip(function_info.parameter_kinds)
656 {
657     if let ValueKind::Indirect = param_kind {
658         let kind_id = Attribute::get_named_enum_kind_id("byval");
659         let byval_attribute = self.context.create_type_attribute(
660             kind_id,
661             param_type
662             .into_pointer_type()
663             .get_element_type()
664             .as_any_type_enum(),
665         );
666         function_value.add_attribute(
667             AttributeLoc::Param(param_index.try_into().unwrap()),
668             byval_attribute,
669         )
670     }
671 }
672 self.function_value_map
673     .borrow_mut()
674     .insert(*function_id, function_value);
675     function_value
676 }
677 }
678
679 struct CodeGenFunctionBodyLoweringContext<
680     'db,
681     'context,
682     'module,
683     'type_cache,
684     'function_info_cache,
685     'function_value_cache,

```

```

686 > {
687     db: &'db dyn CodeGenDatabase,
688     context: &'context Context,
689     module: &'module Module<'context>,
690     type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
691     function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<'db, 'context,
692     ↵ 'type_cache>,
693     function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
694         'db,
695         'context,
696         'module,
697         'type_cache,
698         'function_info_cache,
699     >,
700 }
701 impl<'db, 'context, 'module, 'type_cache, 'function_info_cache, 'function_value_cache>
702     CodeGenFunctionBodyLoweringContext<
703         'db,
704         'context,
705         'module,
706         'type_cache,
707         'function_info_cache,
708         'function_value_cache,
709     >
710 {
711     pub(crate) fn new(
712         db: &'db dyn CodeGenDatabase,
713         context: &'context Context,
714         module: &'module Module<'context>,
715         type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
716         function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<
717             'db,
718             'context,
719             'type_cache,
720         >,
721         function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
722             'db,
723             'context,
724             'module,
725             'type_cache,
726             'function_info_cache,
727         >,
728     ) -> Self {
729         Self {
730             db,
731             context,
732             module,

```

```

733     type_cache,
734     function_info_cache,
735     function_value_cache,
736 }
737 }
738
739 pub(crate) fn build_function_body(&self, function_id: &FunctionDefinitionId) {
740     let llvm_function_value =
741          $\rightarrow$  self.function_value_cache.llvm_function_value(function_id);
742     let llvm_body = self.context.append_basic_block(llvm_function_value,  $"\text{ }$ );
743     let builder = self.context.create_builder();
744     builder.position_at_end(llvm_body);
745     let body = self.db.body_of_definition(*function_id);
746     let function_signature =
747          $\rightarrow$  self.db.callable_definition_signature((*function_id).into());
748     let function_info = self.function_info_cache.function_info(function_id);
749
750     let return_value_ptr = match function_info.return_kind {
751         ValueKind::Indirect => Some(
752             llvm_function_value
753                 .get_first_param()
754                 .unwrap()
755                 .into_pointer_value(),
756         ),
757         ValueKind::Direct => {
758             let ret_type =
759                  $\rightarrow$  llvm_function_value.get_type().get_return_type().unwrap();
760             let bit_size = self.type_cache.target_data().get_bit_size(&ret_type);
761             if bit_size <= 64 {
762                 None
763             } else {
764                 Some(builder.build_alloca(ret_type,  $"\text{ }$ ))
765             }
766         }
767     };
768
769     let parameter_scope = body
770         .parameters
771         .iter()
772         .zip(
773             function_info
774                 .skip_return_param(llvm_function_value.get_param_iter())
775                 .zip(function_info.parameter_kinds)
776                 .zip(function_signature.parameter_types),
777         )
778         .map(|(pattern_id, ((parameter, parameter_kind), param_ty))| {
779             let pattern = &body.patterns[*pattern_id];
780             match pattern {

```

```

778     Pattern::Deconstructor(_, _) => todo!("Using unsupported path
    ↪ pattern"),
779     Pattern::Bind(name) => {
780         parameter.set_name(&name.id);
781         let binding_value = match parameter_kind {
782             ValueKind::Indirect => Value {
783                 kind: parameter_kind,
784                 value: parameter,
785             },
786             ValueKind::Direct => {
787                 if self.type_cache.bit_size(&param_ty) <= 64 {
788                     Value {
789                         kind: ValueKind::Direct,
790                         value: parameter,
791                     }
792                 } else {
793                     let param_ptr =
794                         builder.build_alloca(parameter.get_type(),
795                         ↪ &name.id);
796                     builder.build_store(param_ptr, parameter);
797                     Value {
798                         kind: ValueKind::Indirect,
799                         value: param_ptr.into(),
800                     }
801                 }
802             };
803             Binding {
804                 name: name.clone(),
805                 value: binding_value,
806             }
807         }
808     }
809 }
810 .collect::<Scope>();
811
812 let binding_stack = std::iter::once(parameter_scope).collect();
813
814 let builder = self.context.create_builder();
815 builder.position_at_end(llvm_body);
816
817 let return_value = ExpressionLowerer::new(
818     self.db,
819     &self.context,
820     &builder,
821     &self.type_cache,
822     &self.function_info_cache,
823     &self.function_value_cache,

```

```

824             *function_id,
825             binding_stack,
826         )
827         .fold_expression(return_value_ptr, body.root_expression);
828
829     let return_value = match return_value_ptr {
830         None => {
831             let Value { kind, value } = return_value.unwrap();
832             let value = match kind {
833                 ValueKind::Indirect =>
834                     builder.build_load(value.into_pointer_value(), ""),
835                 ValueKind::Direct => value,
836             };
837             Some(value)
838         }
839         Some(ptr) => match function_info.return_kind {
840             ValueKind::Indirect => None,
841             ValueKind::Direct => Some(builder.build_load(ptr, "")),
842         },
843     };
844     builder.build_return(return_value.as_ref().map(|val| val as &dyn BasicValue));
845 }
846 }
847
848 struct CodeGenFunctionArgumentInfo {
849     kind: ArgumentKind,
850 }
851
852 enum ArgumentKind {
853     Direct,
854     Indirect,
855 }
856
857 #[derive(Clone, Copy)]
858 enum ReturnKind {
859     ArgumentPointer,
860     RegisterValue,
861 }
862
863 #[derive(Debug, Clone, Copy)]
864 enum ValueKind {
865     Indirect,
866     Direct,
867 }
868
869 struct FunctionIr<'ink> {
870     pub value: FunctionValue<'ink>,
```

```

871     pub return_kind: ReturnKind,
872     pub parameters_kind: Vec<ValueKind>,
873 }
874
875 impl FunctionIr<'_> {
876     fn parameters(&self) -> impl Iterator<Item = (BasicValueEnum, &ValueKind)> {
877         match self.return_kind {
878             ReturnKind::ArgumentPointer => self
879                         .value
880                         .get_param_iter()
881                         .zip(self.parameters_kind.iter())
882                         .skip(1),
883             ReturnKind::RegisterValue => self
884                         .value
885                         .get_param_iter()
886                         .zip(self.parameters_kind.iter())
887                         .skip(0),
888         }
889     }
890 }
891
892 pub fn compile_text(source: String, target: CodeGenTarget, liquid: bool) {
893     let mut db = Database::default();
894     db.set_source_file_text(source);
895     db.set_target(target);
896     db.set_liquid(liquid);
897     db.build_assembly_ir()
898 }
899
900 pub fn build_assembly_ir(db: &dyn CodeGenDatabase) {
901     let context = Context::create();
902
903     let target_machine = db.target_machine();
904     let target_data = target_machine.get_target_data();
905
906     let llvm_module = context.create_module("ekitai_module");
907     llvm_module.set_data_layout(&target_data.get_data_layout());
908     llvm_module.set_triple(&target_machine.get_triple());
909
910     let type_cache = CodeGenTypeCache::new(db, &context);
911     let function_info_cache = CodeGenFunctionInfoCache::new(db, &context, &type_cache);
912     let function_value_cache = CodeGenFunctionValueCache::new(
913         db,
914         &context,
915         &llvm_module,
916         &type_cache,
917         &function_info_cache,
918     );

```

```

919
920     let function_body_builder = CodeGenFunctionBodyLoweringContext::new(
921         db,
922         &context,
923         &llvm_module,
924         &type_cache,
925         &function_info_cache,
926         &function_value_cache,
927     );
928
929     let def_map = db.source_file_definitions_map();
930     for function_id in def_map.root_module_item_scope().iter_function_locations() {
931         if db.liquid() {
932             if !hir::liquid::check_abstraction(db.upcast(), *function_id) {
933                 println!("Type error");
934                 return;
935             }
936         }
937         function_body_builder.build_function_body(function_id)
938     }
939
940     println!("{}", llvm_module.print_to_string().to_string());
941     if let Err(err) = llvm_module.verify() {
942         let err = err.to_string();
943         panic!("Could not verify llvm: {}", err);
944     };
945     let _ = llvm_module.print_to_file("out.ll");
946 }
947
948 struct ExpressionLowerer<
949     'db,
950     'context,
951     'module,
952     'builder,
953     'type_cache,
954     'function_info_cache,
955     'function_value_cache,
956     > {
957     db: &'db dyn CodeGenDatabase,
958     context: &'context Context,
959     builder: &'builder Builder<'context>,
960     type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
961     function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<'db, 'context,
962     ↳ 'type_cache>,
963     function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
964         'db,
965         'context,
966         'module,

```

```

966         'type_cache,
967         'function_info_cache,
968     >,
969     function_id: FunctionDefinitionId,
970     body: Body,
971     binding_stack: RefCell<LocalBindingStack<'context>>,
972     inference: InferenceResult,
973 }
974
975 impl<
976     'db,
977     'context,
978     'module,
979     'builder,
980     'type_cache,
981     'function_info_cache,
982     'function_value_cache,
983     >
984     ExpressionLowerer<
985         'db,
986         'context,
987         'module,
988         'builder,
989         'type_cache,
990         'function_info_cache,
991         'function_value_cache,
992     >
993 {
994     pub fn new(
995         db: &'db dyn CodeGenDatabase,
996         context: &'context Context,
997         builder: &'builder Builder<'context>,
998         type_cache: &'type_cache CodeGenTypeCache<'db, 'context>,
999         function_info_cache: &'function_info_cache CodeGenFunctionInfoCache<
1000             'db,
1001             'context,
1002             'type_cache,
1003         >,
1004         function_value_cache: &'function_value_cache CodeGenFunctionValueCache<
1005             'db,
1006             'context,
1007             'module,
1008             'type_cache,
1009             'function_info_cache,
1010         >,
1011         function_id: FunctionDefinitionId,
1012         binding_stack: LocalBindingStack<'context>,
1013     ) -> Self {

```

```
1014
1015     Self {
1016         db,
1017         context,
1018         builder,
1019         type_cache,
1020         function_info_cache,
1021         function_value_cache,
1022         function_id,
1023         body: db.body_of_definition(function_id),
1024         binding_stack: RefCell::new(binding_stack),
1025         inference: db.infer_body_expression_types(function_id),
1026     }
1027 }
1028
1029 pub fn get_owner_function_value(&self) -> FunctionValue {
1030     self.function_value_cache
1031     .llvm_function_value(&self.function_id)
1032 }
1033
1034 fn get_alloca_builder(&self) -> Builder<'context> {
1035     let builder = self.context.create_builder();
1036     let block = self
1037         .get_owner_function_value()
1038         .get_first_basic_block()
1039         .unwrap();
1040     match block.get_first_instruction() {
1041         Some(instruction) => builder.position_before(&instruction),
1042         None => builder.position_at_end(block),
1043     };
1044     builder
1045 }
1046
1047 pub fn fold_expression(
1048     &self,
1049     indirect_value: Option<PointerValue<'context>>,
1050     expr_id: TermId,
1051 ) -> Option<Value<'context>> {
1052     let expr = &self.body.expressions[expr_id];
1053     match expr {
1054         Term::Block {
1055             statements,
1056             trailing_expression,
1057         } => {
1058             for statement in statements {
1059                 match statement {
1060                     Statement::Let(pattern_id, expr_id) => {
1061                         let pattern = &self.body.patterns[*pattern_id];
1062                         let value = self.fold_expression(None, *expr_id).unwrap();
1063                         self.body.expressions[*expr_id] = Value::Let(pattern, value);
1064                     }
1065                 }
1066             }
1067         }
1068     }
1069 }
```

```

1062         let binding = match pattern {
1063             Pattern::Deconstructor(_, _) => todo!(),
1064             Pattern::Bind(name) => Binding {
1065                 name: name.clone(),
1066                 value,
1067             },
1068         };
1069         self.binding_stack
1070             .borrow_mut()
1071             .push(std::iter::once(binding).collect());
1072     }
1073     Statement::Expression(expr_id) => {
1074         self.fold_expression(None, *expr_id);
1075     }
1076 }
1077 }
1078 let value = self.fold_expression(indirect_value, *trailing_expression);
1079 for statement in statements {
1080     if let Statement::Let(_, _) = statement {
1081         self.binding_stack.borrow_mut().pop();
1082     };
1083 }
1084 value
1085 }
1086 Term::If {
1087     condition,
1088     then_branch,
1089     else_branch,
1090 } => self.fold_if_expression(indirect_value, condition, then_branch,
1091                                 else_branch),
1092 Term::Match { matchee, case_list } => {
1093     let resolver =
1094         Resolver::new_for_expression(self.db.upcast(), self.function_id,
1095                                     expr_id);
1096     self.fold_match_expression(indirect_value, *matchee, case_list,
1097                                resolver)
1098 }
1099 Term::Call { callee, arguments } => {
1100     self.fold_call_expression(indirect_value, callee, arguments)
1101 }
1102 Term::Binary(operator, lhs, rhs) => {
1103     self.fold_binary_expression(indirect_value, operator, lhs, rhs)
1104 }
1105 Term::Unary(op, expr) => self.fold_unary_expression(indirect_value, op,
1106                                                        expr),
1107 Term::Path(path) => {
1108     let resolver =

```

```

1105             Resolver::new_for_expression(self.db.upcast(), self.function_id,
1106                                         ↣ expr_id);
1107             self.fold_path_expression(indirect_value, path, resolver)
1108         }
1109         Term::Literal(literal) => {
1110             self.fold_literal_expression(indirect_value, expr_id, literal)
1111         }
1112         Term::New(inner) => self.fold_new_expression(indirect_value, *inner),
1113     }
1114
1115     fn fold_if_expression(
1116         &self,
1117         indirect_value: Option<PointerValue<'context>>,
1118         condition: &TermId,
1119         then_branch: &TermId,
1120         else_branch: &TermId,
1121     ) -> Option<Value<'context>> {
1122         let comparison = self
1123             .fold_expression(None, *condition)
1124             .unwrap()
1125             .value
1126             .into_int_value();
1127
1128         let then_block = self
1129             .context
1130             .append_basic_block(self.get_owener_function_value(), "then");
1131         let else_block = self
1132             .context
1133             .append_basic_block(self.get_owener_function_value(), "else");
1134         let merge_block = self
1135             .context
1136             .append_basic_block(self.get_owener_function_value(), "merge");
1137
1138         self.builder
1139             .build_conditional_branch(comparison, then_block, else_block);
1140
1141         self.builder.position_at_end(then_block);
1142         let then_value = self.fold_expression(indirect_value, *then_branch);
1143         let then_block = self.builder.get_insert_block().unwrap();
1144         self.builder.build_unconditional_branch(merge_block);
1145
1146         self.builder.position_at_end(else_block);
1147         let else_value = self.fold_expression(indirect_value, *else_branch);
1148         let else_block = self.builder.get_insert_block().unwrap();
1149         self.builder.build_unconditional_branch(merge_block);
1150
1151         self.builder.position_at_end(merge_block);

```

```

1152     match indirect_value {
1153         Some(ptr) => None,
1154         None => {
1155             let (
1156                 Value {
1157                     kind: then_kind,
1158                     value: then_value,
1159                 },
1160                 Value {
1161                     kind: else_kind,
1162                     value: else_value,
1163                 },
1164             ) = (then_value.unwrap(), else_value.unwrap());
1165             match (then_kind, else_kind) {
1166                 (ValueKind::Direct, ValueKind::Direct)
1167                 | (ValueKind::Indirect, ValueKind::Indirect) => {
1168                     let phi = self.builder.build_phi(then_value.get_type(), "phi");
1169                     phi.add_incoming(&[(&then_value, then_block), (&else_value,
1170                         ↳ else_block)]);
1171                     Some(Value::new(then_kind, phi.as_basic_value()))
1172                 }
1173                 -> panic!(),
1174             }
1175         }
1176     }
1177
1178     fn fold_match_expression(
1179         &self,
1180         indirect_value: Option<PointerValue<'context>>,
1181         matchee: TermId,
1182         case_list: &[(PatternId, TermId)],
1183         resolver: Resolver,
1184     ) -> Option<Value<'context>> {
1185         let Value {
1186             kind: matchee_kind,
1187             value: matchee_value,
1188         } = self.fold_expression(None, matchee).unwrap();
1189
1190         let matchee_type = &self.inference.type_of_expression[matchee];
1191         let TypeInfo { tag, tag_map } = self.type_cache.type_info(matchee_type);
1192
1193         let tag_value = match matchee_kind {
1194             ValueKind::Indirect => {
1195                 let matchee_ptr = matchee_value.into_pointer_value();
1196                 match tag {
1197                     Some(_) => {

```

```

1198     let tag_ptr = self.builder.build_struct_gep(matchee_ptr, 0,
1199         ↪ "").unwrap();
1200     Some(self.builder.build_load(tag_ptr, "").into_int_value())
1201 }
1202 None => None,
1203 }
1204 ValueKind::Direct => match tag {
1205     Some(_) => Some(
1206         self.builder
1207             .build_extract_value(matchee_value.into_struct_value(), 0, "")
1208             .unwrap()
1209             .into_int_value(),
1210         ),
1211     None => None,
1212 },
1213 };
1214
1215 match tag_value {
1216     Some(tag_value) => {
1217         let (patterns_per_tag_value, else_patterns) = case_list
1218             .iter()
1219             .cloned()
1220             .map(|(pattern_id, expression_id)| {
1221                 let case_pattern = &self.body.patterns[pattern_id];
1222                 match case_pattern {
1223                     Pattern::Deconstructor(path, _) => {
1224                         let constructor_id = resolver
1225                             .resolve_path_in_value_namespace(self.db.upcast(),
1226                                 ↪ path)
1227                             .map(|item| match item {
1228                                 ValueNamespaceItem::ValueConstructor(id) => id,
1229                                 _ => panic!(),
1230                             })
1231                             .unwrap();
1232                         let tag_value = tag_map.get(&constructor_id).unwrap();
1233                         ((pattern_id, expression_id), Some(*tag_value))
1234                     }
1235                     Pattern::Bind(_) => ((pattern_id, expression_id), None),
1236                 }
1237             })
1238             .fold(
1239                 (BTreeMap::new(), Vec::new()),
1240                 |(mut tag_patterns_map, mut else_patterns), (case, tag)| match
1241                     tag {
1242                         None => {
1243                             else_patterns.push(case);
1244                             (tag_patterns_map, else_patterns)
1245                         }
1246                     }
1247                 )
1248             )
1249         }
1250     }
1251 }

```

```

1243                     }
1244                     Some(tag) => {
1245                         tag_patterns_map.entry(tag).or_insert(vec![case]).push(case);
1246                         (tag_patterns_map, else_patterns)
1247                     }
1248                 },
1249             );
1250
1251             let else_block = self
1252                 .context
1253                 .append_basic_block(self.get_owner_function_value(), "case.else");
1254             let merge_block = self
1255                 .context
1256                 .append_basic_block(self.get_owner_function_value(), "case.merge");
1257
1258             let (switch_cases, cases_and_blocks) = patterns_per_tag_value
1259                 .into_iter()
1260                 .map(|(tag, patterns)| {
1261                     let tag_value = tag_value.get_type().const_int(tag as u64,
1262                         ↪ false);
1263                     let case_block = self
1264                         .context
1265                         .prepend_basic_block(merge_block, format!("br.{}.tag",
1266                             ↪ tag).as_str());
1267                         ((tag_value, case_block), (case_block, patterns))
1268                 })
1269                 .unzip::<_, (BasicBlock, Vec<(PatternId, TermId)>), Vec<_>,
1270                         ↪ Vec<_>();
1271
1272             self.builder
1273                 .build_switch(tag_value, else_block,
1274                         ↪ switch_cases.as_slice().as_ref());
1275
1276             self.builder.position_at_end(else_block);
1277             self.builder.build_unreachable();
1278
1279             let case_values = cases_and_blocks
1280                 .into_iter()
1281                 .map(|(case_block, cases)| {
1282                     self.builder.position_at_end(case_block);
1283                     let (pattern_id, expression_id) = cases.first().unwrap();
1284                     let pattern = &self.body.patterns[*pattern_id];
1285                     match pattern {
1286                         Pattern::Deconstructor(_, sub_patterns) => {
1287                             if sub_patterns.is_empty() {
1288                                 let case_value =
1289                                     self.fold_expression(indirect_value,
1290                                         ↪ *expression_id);
1291
1292                                 self.builder.position_at_end(case_value);
1293
1294                                 self.builder.append(case_block);
1295
1296                                 self.builder.append(case_value);
1297
1298                                 self.builder.append(case_block);
1299
1300                                 self.builder.append(case_value);
1301
1302                                 self.builder.append(case_block);
1303
1304                                 self.builder.append(case_value);
1305
1306                                 self.builder.append(case_block);
1307
1308                                 self.builder.append(case_value);
1309
1310                                 self.builder.append(case_block);
1311
1312                                 self.builder.append(case_value);
1313
1314                                 self.builder.append(case_block);
1315
1316                                 self.builder.append(case_value);
1317
1318                                 self.builder.append(case_block);
1319
1320                                 self.builder.append(case_value);
1321
1322                                 self.builder.append(case_block);
1323
1324                                 self.builder.append(case_value);
1325
1326                                 self.builder.append(case_block);
1327
1328                                 self.builder.append(case_value);
1329
1330                                 self.builder.append(case_block);
1331
1332                                 self.builder.append(case_value);
1333
1334                                 self.builder.append(case_block);
1335
1336                                 self.builder.append(case_value);
1337
1338                                 self.builder.append(case_block);
1339
1340                                 self.builder.append(case_value);
1341
1342                                 self.builder.append(case_block);
1343
1344                                 self.builder.append(case_value);
1345
1346                                 self.builder.append(case_block);
1347
1348                                 self.builder.append(case_value);
1349
1350                                 self.builder.append(case_block);
1351
1352                                 self.builder.append(case_value);
1353
1354                                 self.builder.append(case_block);
1355
1356                                 self.builder.append(case_value);
1357
1358                                 self.builder.append(case_block);
1359
1360                                 self.builder.append(case_value);
1361
1362                                 self.builder.append(case_block);
1363
1364                                 self.builder.append(case_value);
1365
1366                                 self.builder.append(case_block);
1367
1368                                 self.builder.append(case_value);
1369
1370                                 self.builder.append(case_block);
1371
1372                                 self.builder.append(case_value);
1373
1374                                 self.builder.append(case_block);
1375
1376                                 self.builder.append(case_value);
1377
1378                                 self.builder.append(case_block);
1379
1380                                 self.builder.append(case_value);
1381
1382                                 self.builder.append(case_block);
1383
1384                                 self.builder.append(case_value);
1385
1386                                 self.builder.append(case_block);
1387
1388                                 self.builder.append(case_value);
1389
1390                                 self.builder.append(case_block);
1391
1392                                 self.builder.append(case_value);
1393
1394                                 self.builder.append(case_block);
1395
1396                                 self.builder.append(case_value);
1397
1398                                 self.builder.append(case_block);
1399
1400                                 self.builder.append(case_value);
1401
1402                                 self.builder.append(case_block);
1403
1404                                 self.builder.append(case_value);
1405
1406                                 self.builder.append(case_block);
1407
1408                                 self.builder.append(case_value);
1409
1410                                 self.builder.append(case_block);
1411
1412                                 self.builder.append(case_value);
1413
1414                                 self.builder.append(case_block);
1415
1416                                 self.builder.append(case_value);
1417
1418                                 self.builder.append(case_block);
1419
1420                                 self.builder.append(case_value);
1421
1422                                 self.builder.append(case_block);
1423
1424                                 self.builder.append(case_value);
1425
1426                                 self.builder.append(case_block);
1427
1428                                 self.builder.append(case_value);
1429
1430                                 self.builder.append(case_block);
1431
1432                                 self.builder.append(case_value);
1433
1434                                 self.builder.append(case_block);
1435
1436                                 self.builder.append(case_value);
1437
1438                                 self.builder.append(case_block);
1439
1440                                 self.builder.append(case_value);
1441
1442                                 self.builder.append(case_block);
1443
1444                                 self.builder.append(case_value);
1445
1446                                 self.builder.append(case_block);
1447
1448                                 self.builder.append(case_value);
1449
1450                                 self.builder.append(case_block);
1451
1452                                 self.builder.append(case_value);
1453
1454                                 self.builder.append(case_block);
1455
1456                                 self.builder.append(case_value);
1457
1458                                 self.builder.append(case_block);
1459
1460                                 self.builder.append(case_value);
1461
1462                                 self.builder.append(case_block);
1463
1464                                 self.builder.append(case_value);
1465
1466                                 self.builder.append(case_block);
1467
1468                                 self.builder.append(case_value);
1469
1470                                 self.builder.append(case_block);
1471
1472                                 self.builder.append(case_value);
1473
1474                                 self.builder.append(case_block);
1475
1476                                 self.builder.append(case_value);
1477
1478                                 self.builder.append(case_block);
1479
1480                                 self.builder.append(case_value);
1481
1482                                 self.builder.append(case_block);
1483
1484                                 self.builder.append(case_value);
1485
1486                                 self.builder.append(case_block);
1487
1488                                 self.builder.append(case_value);
1489
1490                                 self.builder.append(case_block);
1491
1492                                 self.builder.append(case_value);
1493
1494                                 self.builder.append(case_block);
1495
1496                                 self.builder.append(case_value);
1497
1498                                 self.builder.append(case_block);
1499
1500                                 self.builder.append(case_value);
1501
1502                                 self.builder.append(case_block);
1503
1504                                 self.builder.append(case_value);
1505
1506                                 self.builder.append(case_block);
1507
1508                                 self.builder.append(case_value);
1509
1510                                 self.builder.append(case_block);
1511
1512                                 self.builder.append(case_value);
1513
1514                                 self.builder.append(case_block);
1515
1516                                 self.builder.append(case_value);
1517
1518                                 self.builder.append(case_block);
1519
1520                                 self.builder.append(case_value);
1521
1522                                 self.builder.append(case_block);
1523
1524                                 self.builder.append(case_value);
1525
1526                                 self.builder.append(case_block);
1527
1528                                 self.builder.append(case_value);
1529
1530                                 self.builder.append(case_block);
1531
1532                                 self.builder.append(case_value);
1533
1534                                 self.builder.append(case_block);
1535
1536                                 self.builder.append(case_value);
1537
1538                                 self.builder.append(case_block);
1539
1540                                 self.builder.append(case_value);
1541
1542                                 self.builder.append(case_block);
1543
1544                                 self.builder.append(case_value);
1545
1546                                 self.builder.append(case_block);
1547
1548                                 self.builder.append(case_value);
1549
1550                                 self.builder.append(case_block);
1551
1552                                 self.builder.append(case_value);
1553
1554                                 self.builder.append(case_block);
1555
1556                                 self.builder.append(case_value);
1557
1558                                 self.builder.append(case_block);
1559
1560                                 self.builder.append(case_value);
1561
1562                                 self.builder.append(case_block);
1563
1564                                 self.builder.append(case_value);
1565
1566                                 self.builder.append(case_block);
1567
1568                                 self.builder.append(case_value);
1569
1570                                 self.builder.append(case_block);
1571
1572                                 self.builder.append(case_value);
1573
1574                                 self.builder.append(case_block);
1575
1576                                 self.builder.append(case_value);
1577
1578                                 self.builder.append(case_block);
1579
1580                                 self.builder.append(case_value);
1581
1582                                 self.builder.append(case_block);
1583
1584                                 self.builder.append(case_value);
1585
1586                                 self.builder.append(case_block);
1587
1588                                 self.builder.append(case_value);
1589
1590                                 self.builder.append(case_block);
1591
1592                                 self.builder.append(case_value);
1593
1594                                 self.builder.append(case_block);
1595
1596                                 self.builder.append(case_value);
1597
1598                                 self.builder.append(case_block);
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2898
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2938
2939
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2948
2949
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3089
3090
3091
3092
3
```

```
1286         self.builder.build_unconditional_branch(merge_block);
1287
1288         let case_block =
1289             ↪ self.builder.get_insert_block().unwrap();
1290             (case_value, case_block)
1291     } else {
1292         let inner_value = match matchee_kind {
1293             ValueKind::Indirect => {
1294                 let inner_ptr = self
1295                     .builder
1296                     .build_struct_gep(
1297                         matchee_value.into_pointer_value(),
1298                         1,
1299                         ""),
1300                     )
1301                     .unwrap();
1302                     Value::new(
1303                         ValueKind::Indirect,
1304                         inner_ptr.as_basic_value_enum(),
1305                     )
1306             }
1307             ValueKind::Direct => {
1308                 let inner_value = self
1309                     .builder
1310                     .build_extract_value(
1311                         matchee_value.into_struct_value(),
1312                         1,
1313                         ""),
1314                     )
1315                     .unwrap();
1316                     Value::new(ValueKind::Direct, inner_value)
1317             }
1318         };
1319
1320         let scope = sub_patterns
1321             .iter()
1322             .enumerate()
1323             .map(|(index, sub_pattern_id)| {
1324                 let sub_pattern =
1325                     &self.body.patterns[*sub_pattern_id];
1326                 match sub_pattern {
1327                     Pattern::Bind(name) => match
1328                         inner_value.kind {
1329                             ValueKind::Indirect => {
1330                                 let bind_ptr = self
1331                                     .builder
1332                                     .build_struct_gep(
1333                                         inner_value
1334                                         .value
```

```

1331         .into_pointer_value(),
1332         index as u32,
1333         """",
1334     )
1335     .unwrap();
1336     let value = if self
1337         .type_cache
1338         .target_data()
1339         .get_bit_size(
1340             &bind_ptr
1341                 .get_type()
1342                 .get_element_type(),
1343             )
1344             <= 64
1345     {
1346         Value::new(
1347             ValueKind::Direct,
1348             self.builder
1349             .build_load(bind_ptr, """),
1350             )
1351     } else {
1352         Value::new(
1353             ValueKind::Indirect,
1354             bind_ptr.as_basic_value_enum(),
1355             )
1356     };
1357     Binding {
1358         name: name.clone(),
1359         value,
1360     }
1361 }
1362 ValueKind::Direct => {
1363     let value = self
1364         .builder
1365         .build_extract_value(
1366             inner_value
1367                 .value
1368                 .into_struct_value(),
1369                 index as u32,
1370                 """",
1371             )
1372         .unwrap();
1373     Binding {
1374         name: name.clone(),
1375         value: Value {
1376             kind: ValueKind::Direct,
1377             value,
1378         },

```

```

1379                         }
1380                     }
1381                 },
1382             Pattern::Deconstructor(_, _) => todo!(),
1383         }
1384     })
1385     .collect();
1386
1387     self.binding_stack.borrow_mut().push(scope);
1388     let case_value =
1389         self.fold_expression(indirect_value,
1390             ↪ *expression_id);
1391     self.builder.build_unconditional_branch(merge_block);
1392     self.binding_stack.borrow_mut().pop();
1393     let case_block =
1394         ↪ self.builder.get_insert_block().unwrap();
1395     (case_value, case_block)
1396     }
1397   }
1398 }
1399 .collect::<Vec<_>>();
1400
1401 self.builder.position_at_end(merge_block);
1402 match indirect_value {
1403   Some(_) => None,
1404   None => {
1405     let first_value =
1406       ↪ case_values.first().unwrap().0.as_ref().unwrap();
1407     let phi = self.builder.build_phi(first_value.value.get_type(),
1408         ↪ "phi");
1409     phi.add_incoming(
1410       case_values
1411         .iter()
1412         .map(|(case_value, case_block)| {
1413           (
1414             &case_value.as_ref().unwrap().value as &dyn
1415             ↪ BasicValue,
1416             *case_block,
1417           )
1418         })
1419         .collect::<Vec<_>>()
1420         .as_slice(),
1421     );
1422     Some(Value::new(first_value.kind, phi.as_basic_value()))
1423   }

```

```

1422         }
1423     }
1424
1425     None => todo!(),
1426 }
1427 }
1428
1429 fn fold_call_expression(
1430     &self,
1431     indirect_value: Option<PointerValue<'context>>,
1432     callee: &TermId,
1433     arguments: &[TermId],
1434 ) -> Option<Value<'context>> {
1435     let callee_type = &self.inference.type_of_expression[*callee];
1436     let callable_definition = match callee_type {
1437         Type::FunctionDefinition(callable) => callable,
1438         _ => panic!("call has no callable type."),
1439     };
1440     match callable_definition {
1441         CallableDefinitionId::FunctionDefinition(id) => {
1442             let function_info = self.function_info_cache.function_info(&id);
1443             let function_value = self.function_value_cache.llvm_function_value(&id);
1444
1445             let return_type = function_info.get_return_type(function_value);
1446
1447             let indirect_function_return = match function_info.return_kind {
1448                 ValueKind::Direct => None,
1449                 ValueKind::Indirect => match indirect_value {
1450                     Some(ptr) => Some(ptr.as_basic_value_enum()),
1451                     None => Some(
1452                         self.get_alloca_builder()
1453                         .build_alloca(
1454                             BasicTypeEnum::try_from(
1455                                 return_type.into_pointer_type().get_element_type(),
1456                             )
1457                             .unwrap(),
1458                             ""),
1459                         )
1460                         .as_basic_value_enum(),
1461                         ),
1462                     },
1463                 };
1464
1465             let arguments = indirect_function_return.into_iter().chain(
1466                 arguments
1467                 .iter()
1468                 .zip(function_info.parameter_kinds.iter())
1469                 .map(|(argument_expr, parameter_kind)| {

```

```

1470         let Value {
1471             kind: argument_kind,
1472             value: argument_value,
1473         } = self.fold_expression(None, *argument_expr).unwrap();
1474         let argument_value = match (parameter_kind, argument_kind) {
1475             (ValueKind::Indirect, ValueKind::Indirect)
1476             | (ValueKind::Direct, ValueKind::Direct) =>
1477                 argument_value,
1478             (ValueKind::Direct, ValueKind::Indirect) => self
1479                 .builder
1480                 .build_load(argument_value.into_pointer_value(), ""),
1481             (ValueKind::Indirect, ValueKind::Direct) => {
1482                 let ptr = self
1483                     .get_alloca_builder()
1484                     .build_alloca(argument_value.get_type(), "");
1485                 self.builder.build_store(ptr, argument_value);
1486                 ptr.as_basic_value_enum()
1487             }
1488         };
1489         argument_value
1490     },
1491
1492     let call_value = self.builder.build_call(
1493         function_value,
1494         arguments.map(|x| x.into()).collect::<Vec<_>>().as_slice(),
1495         ""),
1496     );
1497
1498     match indirect_value {
1499         Some(_) => None,
1500         None => match function_info.return_kind {
1501             ValueKind::Indirect => Some(Value::new(
1502                 ValueKind::Indirect,
1503                 indirect_function_return.unwrap(),
1504             )),
1505             ValueKind::Direct => Some(Value::new(
1506                 ValueKind::Direct,
1507                 call_value.try_as_basic_value().unwrap_left(),
1508             )),
1509         },
1510     }
1511 }
1512 CallableDefinitionId::ValueConstructor(constructor_id) => {
1513     let (struct_type, type_info) = self
1514         .type_cache
1515         .adt_struct(constructor_id.type_definition_id);
1516     match indirect_value {

```

```

1517     Some(ptr) => {
1518         if let Some(tag_type) = type_info.tag {
1519             let tag_value = type_info.tag_map[&constructor_id];
1520             let tag_value = tag_type.const_int(tag_value as u64, false);
1521             let tag_ptr = self.builder.build_struct_gep(ptr, 0,
1522                 & "").unwrap();
1523             self.builder.build_store(tag_ptr, tag_value);
1524             let variant_ptr = self.builder.build_struct_gep(ptr, 1,
1525                 & "").unwrap();
1526             let variant_ptr = self
1527                 .builder
1528                 .build_bitcast(
1529                     variant_ptr,
1530                     self.type_cache
1531                         .value_constructor_struct(*constructor_id)
1532                         .ptr_type(AddressSpace::Generic),
1533                         ""),
1534                 )
1535                 .into_pointer_value();
1536             let arguments = arguments
1537                 .iter()
1538                 .map(|argument| self.fold_expression(None,
1539                     & *argument).unwrap());
1540             for (index, argument) in arguments.enumerate() {
1541                 let argument_ptr = self
1542                     .builder
1543                     .build_struct_gep(variant_ptr, index as u32, "")
1544                     .unwrap();
1545                 let value = match argument.kind {
1546                     ValueKind::Indirect => {
1547                         if !argument_ptr
1548                             .get_type()
1549                             .get_element_type()
1550                             .is_pointer_type()
1551                             {
1552                                 self.builder
1553                                     .build_load(argument.value.into_pointer_value(),
1554                                         & "")
1555                                     } else {
1556                                         argument.value
1557                                         }
1558                                     }
1559                     ValueKind::Direct => argument.value,
1560                 };
1561             self.builder.build_store(argument_ptr, value);
1562         }
1563     } else {
1564         todo!()
1565     }
1566 }
```

```

1561         }
1562         None
1563     }
1564     None => {
1565         let type_size = self
1566             .type_cache
1567             .target_data()
1568             .get_bit_size(&struct_type.as_any_type_enum());
1569
1570         let value = struct_type.const_zero();
1571         let TypeInfo { tag, tag_map } = type_info;
1572         let tag_value = tag.map(|tag_type| {
1573             tag_type.const_int(tag_map[&constructor_id] as u64, true)
1574         });
1575         let value = if let Some(tag_value) = tag_value {
1576             if type_size <= 64 {
1577                 let value = self
1578                     .builder
1579                     .build_insert_value(value, tag_value, 0, "")
1580                     .unwrap()
1581                     .as_basic_value_enum();
1582             Value::new(ValueKind::Direct, value)
1583         } else {
1584             let ptr = self.get_alloca_builder().build_alloca(struct_type,
1585                 "");
1586             self.builder.build_store(
1587                 ptr,
1588                 ptr.get_type()
1589                     .get_element_type()
1590                     .into_struct_type()
1591                     .const_zero(),
1592             );
1593             let tag_ptr = self.builder.build_struct_gep(ptr, 0,
1594                 "").unwrap();
1595             self.builder.build_store(tag_ptr, tag_value);
1596             let variant_ptr =
1597                 self.builder.build_struct_gep(ptr, 1, "").unwrap();
1598             let variant_ptr = self
1599                 .builder
1600                 .build_bitcast(
1601                     variant_ptr,
1602                     self.type_cache
1603                         .value_constructor_struct(*constructor_id)
1604                         .ptr_type(AddressSpace::Generic),
1605                         "",
1606                 )
1607                 .into_pointer_value();
1608             let arguments = arguments

```

```

1607         .iter()
1608         .map(|argument| self.fold_expression(None,
1609             ↪ *argument).unwrap()));
1610         for (index, argument) in arguments.enumerate() {
1611             let argument_ptr = self
1612                 .builder
1613                     .build_struct_gep(variant_ptr, index as u32, "")
1614                     .unwrap();
1615             let value = match argument.kind {
1616                 ValueKind::Indirect => self
1617                     .builder
1618                         .build_load(argument.value.into_pointer_value(),
1619                             ↪ ""),
1620                         ValueKind::Direct => argument.value,
1621             };
1622             self.builder.build_store(argument_ptr, value);
1623         }
1624     } else {
1625         todo!()
1626     };
1627     Some(value)
1628 }
1629 }
1630 }
1631 }
1632 }
1633
1634 fn fold_binary_expression(
1635     &self,
1636     indirect_value: Option<PointerValue<'context>>,
1637     operator: &BinaryOperator,
1638     lhs: &TermId,
1639     rhs: &TermId,
1640 ) -> Option<Value<'context>> {
1641     let lhs = self
1642         .fold_expression(None, *lhs)
1643         .unwrap()
1644         .value
1645         .into_int_value();
1646     let rhs = self
1647         .fold_expression(None, *rhs)
1648         .unwrap()
1649         .value
1650         .into_int_value();
1651     let int_value = match operator {

```

```

1652     BinaryOperator::Arithmetric(arithmetric_op) => match arithmetric_op {
1653         ArithmeticOperator::Add => self.builder.build_int_add(lhs, rhs, ""),
1654         ArithmeticOperator::Sub => self.builder.build_int_sub(lhs, rhs, ""),
1655         ArithmeticOperator::Div => self.builder.build_int_signed_div(lhs, rhs,
1656             ↪ ""),
1657         ArithmeticOperator::Mul => self.builder.build_int_mul(lhs, rhs, ""),
1658         ArithmeticOperator::Rem => self.builder.build_int_signed_rem(lhs, rhs,
1659             ↪ ""),
1660     },
1661     BinaryOperator::Compare(compare_op) => {
1662         let predicate = match compare_op {
1663             CompareOperator::Equality { negated } => match negated {
1664                 true => IntPredicate::NE,
1665                 false => IntPredicate::EQ,
1666             },
1667             CompareOperator::Order { ordering, strict } => match (ordering,
1668                 ↪ strict) {
1669                 (Ordering::Less, true) => IntPredicate::SLT,
1670                 (Ordering::Less, false) => IntPredicate::SLE,
1671                 (Ordering::Greater, true) => IntPredicate::SGT,
1672                 (Ordering::Greater, false) => IntPredicate::SGE,
1673             },
1674         };
1675         self.builder.build_int_compare(predicate, lhs, rhs, "")
1676     }
1677     BinaryOperator::Logic(logic_op) => match logic_op {
1678         LogicOperator::And => self.builder.build_and(lhs, rhs, ""),
1679         LogicOperator::Or => self.builder.build_or(lhs, rhs, ""),
1680     },
1681     match indirect_value {
1682         Some(ptr) => {
1683             self.builder
1684                 .build_store(ptr, int_value.as_basic_value_enum());
1685             None
1686         }
1687     }
1688
1689     fn fold_unary_expression(
1690         &self,
1691         indirect_value: Option<PointerValue<'context>>,
1692         operator: &UnaryOperator,
1693         expression: &TermId,
1694     ) -> Option<Value<'context>> {
1695         let Value { value: expr, kind } = self.fold_expression(None,
1696             ↪ *expression).unwrap();

```

```

1696     let (kind, value) = match operator {
1697         UnaryOperator::Minus => (
1698             kind,
1699             self.builder
1700                 .build_int_sub(
1701                     expr.get_type().into_int_type().const_zero(),
1702                     expr.into_int_value(),
1703                     ""),
1704                 )
1705                 .into(),
1706             ),
1707         UnaryOperator::Negation => (
1708             kind,
1709             self.builder.build_not(expr.into_int_value(), "").into(),
1710         ),
1711         UnaryOperator::Reference => {
1712             todo!("reference operator lowering to llvm is not implemented")
1713         }
1714         UnaryOperator::Dereference => match kind {
1715             ValueKind::Indirect => {
1716                 expr.into_pointer_value();
1717                 (kind, expr)
1718             }
1719             ValueKind::Direct => (ValueKind::Indirect, expr),
1720         },
1721     };
1722     match indirect_value {
1723         Some(ptr) => {
1724             self.builder.build_store(ptr, value);
1725             None
1726         }
1727         None => Some(Value::new(kind, value)),
1728     }
1729 }
1730
1731 fn fold_path_expression(
1732     &self,
1733     indirect_value: Option<PointerValue<'context>>,
1734     path: &Path,
1735     resolver: Resolver,
1736 ) -> Option<Value<'context>> {
1737     let item = resolver
1738         .resolve_path_in_value_namespace(self.db.upcast(), path)
1739         .unwrap();
1740
1741     match item {
1742         ValueNamespaceItem::Function(_) => todo!(),
1743         ValueNamespaceItem::ValueConstructor(_) => todo!(),

```

```

1744     ValueNamespaceItem::LocalBinding(pattern_id) => match
1745       &self.body.patterns[pattern_id] {
1746         Pattern::Deconstructor(_, _) => todo!(),
1747         Pattern::Bind(name) => {
1748           let value = self
1749             .binding_stack
1750               .borrow()
1751               .get(name)
1752               .expect("missing id {name:?} from scope.");
1753             match indirect_value {
1754               Some(ptr) => {
1755                 match value.kind {
1756                   ValueKind::Indirect => {
1757                     let source = value.value.into_pointer_value();
1758                     let _ = self.builder.build_memcpy(
1759                       ptr,
1760                       4,
1761                       source,
1762                       4,
1763                       self.context.i8_type().const_int(16, false),
1764                     );
1765                     ValueKind::Direct => panic!("trying to put direct
1766                                     value into indirect value at bind {name:?}."),
1767                   }
1768                   None =>
1769                     Some(value.clone()),
1770                   }
1771                 }
1772               },
1773             }
1774           }
1775
1776         fn fold_literal_expression(
1777           &self,
1778           indirect_value: Option<PointerValue<'context>>,
1779           literal_id: TermId,
1780           literal: &Literal,
1781         ) -> Option<Value<'context>> {
1782           let literal_type = &self.inference.type_of_expression[literal_id];
1783           let value = match literal_type {
1784             Type::AbstractDataType(_) => todo!(),
1785             Type::FunctionDefinition(_) => todo!(),
1786             Type::Scalar(ScalarType::Integer(kind)) => {
1787               let value = match literal {
1788                 Literal::Integer(value, _) => *value,
1789                 _ => panic!(),

```

```

1790     };
1791     match kind {
1792         IntegerKind::I32 => {
1793             self.context.i32_type().const_int(value as u64, true).into()
1794         }
1795         IntegerKind::I64 => {
1796             self.context.i64_type().const_int(value as u64, true).into()
1797         }
1798     }
1799 }
1800 Type::Scalar(ScalarType::Boolean) => match literal {
1801     Literal::Bool(bool) => match bool {
1802         true => self.context.bool_type().const_all_ones().into(),
1803         false => self.context.bool_type().const_zero().into(),
1804         _,
1805     } => panic!(),
1806 },
1807     Type::Pointer(_) => todo!(),
1808 };
1809 match indirect_value {
1810     Some(ptr) => {
1811         self.builder.build_store(ptr, value);
1812         None
1813     }
1814     None => Some(Value::new(ValueKind::Direct, value)),
1815 }
1816 }
1817
1818 fn fold_new_expression(
1819     &self,
1820     indirect_value: Option<PointerValue<'context>>,
1821     expr: TermId,
1822 ) -> Option<Value<'context>> {
1823     let ty = &self.inference.type_of_expression[expr];
1824     let ty = self.type_cache.llvm_type(ty);
1825     let ptr = self.builder.build_malloc(ty, "").unwrap();
1826     let _ = self.fold_expression(Some(ptr), expr).is_none();
1827     match indirect_value {
1828         Some(return_ptr) => {
1829             let i8_ptr_type =
1830                 self.context.i8_type().ptr_type(AddressSpace::Generic);
1831             let source = self
1832                 .builder
1833                 .build_bitcast(ptr, i8_ptr_type, "")
1834                 .into_pointer_value();
1835             let source_type = &ptr.get_type().get_element_type().as_any_type_enum();
1836             let source_alignment =
1837                 self.type_cache.target_data().get_abi_alignment(source_type);

```

```
1836     let sink = self
1837         .builder
1838         .build_bitcast(return_ptr, i8_ptr_type, "")
1839         .into_pointer_value();
1840     let sink_alignment = self.type_cache.target_data().get_abi_alignment(
1841         &return_ptr.get_type().get_element_type().as_any_type_enum(),
1842     );
1843     let size = self.type_cache.target_data().get_abi_size(source_type);
1844     let size = self
1845         .context
1846         .ptr_sized_int_type(self.type_cache.target_data(), None)
1847         .const_int(size, false);
1848     let _ =
1849         self.builder
1850             .build_memcpy(sink, sink_alignment, source, source_alignment,
1851                           size);
1852     None
1853 }
1854 None => Some(Value::new(ValueKind::Direct, ptr.as_basic_value_enum())),
1855 }
1856 }
```

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/challenging-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.