RISC

RESEARCH INSTITUTE FOR SYMBOLIC COMPUTATION

JOHANNES KEPLER UNIVERSITY LINZ

Refinement Types for Elm

Lucas Payr

April 2021

RISC Report Series No. 21-10

Available at https://doi.org/10.35011/risc.21-10

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Lucas Payr
K01556372

Submitted at
Research Institute for Symbolic Computation

Supervisor
A.Univ.-Prof. DI Dr.
Wolfgang Schreiner

April 2021

REFINEMENT TYPES FOR ELM

Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Mathematics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Zusammenfassung

Das Ziel dieser Arbeit ist es, das Typsystem von Elm um sogenannte “Refinement” Typen zu erweitern. Elm ist eine rein funktionale Programmiersprache, welche das Hindley-Miler Typsystem benutzt. Refinement Typen sind eine Form von Subtypen, welche anhand eines Prädikats ihre enthaltenen Werte bestimmen. Eine solche Klasse von Refinement Typen, welche für Hindely-Miler Typsysteme ausgelegt ist, sind die sogenannten “Liquid” Typen. Für sie existiert ein Algorithmus, um für einen Ausdruck den dazugehörigen Typ herzuleiten. Dieser Algorithmus interagiert mit einem SMT Solver, um bestimmte Bedingungen für die Subtypen zu erfüllen. Typsysteme, welche mit Liquid Typen erweitert werden, sind nicht länger vollständig, d.h. sie können nur für bestimmte Ausdrücke hergeleitet werden. Die Prädikate sind ebenfalls restringiert. Diese Arbeit liefert eine formale Definition von Elm und dessen Typsystem, auf deren Basis das System mit Liquid Typen erweitert wird. Hierfür wird eine Teilmenge der Ausdrücke sowie eine Teilmenge der Prädikate präsentiert, um die Wohldefiniertheit der Liquid Typen zu gewährleisten. Zum Überprüfen unserer Ergebnisse benützen wir das freie Softwaresystem “K Framework” sowie eine Implementierung in Elm des Algorithmus zum Lösen der Bedingungen für Subtypen. Als SMT Solver benutzen wir Microsofts Software Z3.

Abstract

The aim of this thesis is to add refinement types to Elm. Elm is a pure functional programming language that uses a Hindley-Miler type system. Refinement types are subtypes of existing types. These subtypes are defined by a predicate that specifies which values are part of the subtypes. To extend a Hindley-Miler type system, one can use so-called liquid types. These are special refinement types that come with an algorithm for type inference. This algorithm interacts with an SMT solver to solve subtyping conditions. A type system using liquid types is not complete, meaning not every valid program can be checked. Instead, liquid types are only defined for a subset of expressions and only allow specific predicates. In this thesis, we give a formal definition of the Elm language and its type system. We extend the type system with liquid types and provide a subset of expressions and a subset of predicates such that the extended type system is sound. We use the free software system "K Framework" for rapid prototyping of the formal Elm type system. For rapid prototyping of the core algorithm of the extended type system we implemented said algorithm in Elm and checked the subtyping condition in Microsoft's SMT solver Z3.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

.....

Lucas Payr

Thanks

I would like to thank my supervisor Wolfgang Schreiner for his continuous advice and suggestions throughout numerous revisions. Your input helped a lot.

I would also like to thank my friends and my girlfriend for this lovely time I had at university. I would not have been able to finish my degree if it would not have been without your love and support. It was a great time.

Last but not least, a big thanks to my family. I am privileged to have you all.

Contents

1. Introduction

On 21. September 1997, the onboard computer of the USS Yorktown aircraft carrier threw an uncaught division by zero exception. This resulted in the computer shutting down and the ship becoming unable to be controlled until an engineer was able to restart the computer. Fortunately this happened during a training maneuver [Par19].

Typically, such errors can only be found by extensive testing. Instead one might try to use a more expressive type system that can ensure at compile time that division by zero and similar bugs are impossible to occur.

These more expressive types are called *Refinement Types* [FP91]. Some authors also call them *Dependent Types* [Chr18], though dependent types are typically more general: They are used to prove specific properties by letting the type definition depend on a quantified predicate, whereas a refinement type takes an existing type and excludes certain values that do not ensure a specific property. To avoid confusion, we will use the term "refinement types" within this thesis. The predicate describing the valid values of a refinement type is called a *refinement*.

Refinement types were first introduced by Tim Freeman and Frank Pfenning in 1991 [FP91]. In 2008, Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala from the university of California came up with a variant of refinement types called Logically Quantified Data types, or *Liquid* types for short. Liquid types limit themself to refinements on Integers and Booleans written in propositional logic together with order relations and addition.

Most work on liquid types was done in the UCSD Programming System research group, from which the original paper originated. This group has presented different implementations of liquid types:

- **DSolve** for OCaml/ML [KRJ10]. This type checker originated from the original paper, where liquid types could only be established for a calculus called λ_L. DSolve first translates OCaml to λ_L and then checks the result for type safety.
- **CSolve** for C [Ron+12]. As a follow-up to DSolve, this checker implements Low-Level Liquid Type (LTLL) [RKJ10] for a formal language called *NanoC* that extends λ_L with pointer arithmetic.

- **LiquidHaskell** for Haskell [Vaz+14]. Extending λ_L, this type checker uses a new calculus called λ_P, a polymorphic λ-calculus [VRJ13]. Newer versions can also reason about termination, totality (of functions) and basic theorems.
- **RefScript** for TypeScript [VCJ16]. In a two phase process, the dynamic typed language gets translated into a static typed abstract language [VCJ15]. This language can then be type checked using λ_P.

Outside of that research group, refinement types have also been implemented for Racket [KKT16], Ruby [Kaz+18] and F# [Ben+08].

Elm is a pure functional programming language invented in 2012 by Evan Czaplicki in his master thesis [CC13]. Elm has many similarities to Haskell but is simpler in nature. Its programs are based on a special architecture similar to state machines instead of the monads used in Haskell. This architecture ensures that no runtime error can occur: Errors are modelled via types and therefore need to be handled at compile time. The language compiles to JavaScript which gives the Elm community the unique position of being the first contact with functional programming for many web developers.

This unique position gave rise to design philosophies for writing good functional code, not only for Elm. One such philosophy is "Make impossible states impossible": Model your program in such a way that any possible state of the model represents a valid state of the program. Refinement types provide a way to achieve that rule for integers. Even simple types like a subtype containing all natural numbers or a type containing a range of numbers would be of help.

The goal of this thesis is therefore to define Liquid Types for Elm. In particular, we define a set of predicates $\mathcal{Q}$ such that the type of ever Elm program with if-conditions in $\mathcal{Q}$ can be inferred. $\mathcal{Q}$ needs to allow the definition of range types, natural numbers, and non-zero integers.

Elm has changed a lot since 2012 and therefore the formal model described in the original thesis is outdated. We have started our work by formally defining the Elm language. This includes the syntax, denotational semantic, types and a system of inference rules to infer them. We then introduce the formal notion of liquid types to our type system. We also define the subset of allowed predicates in an if-condition and extend the syntax with refinement types. Furthermore, we present altered inference rules to ensure that liquid types can be defined. Here we introduce the notion of subtyping conditions. We describe an algorithm to solve these conditions and specify a set of predicates that may be used for deriving a refinement solving the subtyping conditions. The algorithm generates SMT statements that can be checked using an

SMT solver. To demonstrate everything in tandem, we provide an implementation of the subtyping algorithm using Z3[MB08] for solving the SMT statements.

The remaining thesis is structured as follows: In Chapter 2, we give a quick history of type systems and the Elm language. In Chapter 3, we formally define the type system of Elm. In Chapter 4, we introduce refinement types and describe how we can extend the type system using liquid types. In Chapter 5, we discuss the implementation and provide a demonstration. In Chapter 6, we evaluate our results and present our conclusions.

2. State of the Art

In this chapter we give a quick history of type theory and the Elm language.

2.1. Type Theory

In 1902, Bertrand Russell wrote Gottlob Frege a letter, pointing out a paradox in Gottlob's definition of sets in his paper *Begriffsschrift* [Fre84]. Up to this point, mathematicians expected that the naive theory of sets was well-formed. Russell's paradox gave a contradiction to this assumption:

Theorem 1.1: Russell's Paradox

Given the set of all sets that do not contain themselfs $R = \{x | x \notin x\}$, then $R \in R$ and $R \notin R$.

To fix this problem, Russell added a basic theory of types in the appendix of *Principia Mathematica* [WR27], which at the time was already in the printer. This rushed theory was not perfect, but it marks the first appearance.

Russell thought that the main pain point of set theory was that sets could be defined implicitly (without listing all elements). Type theory is therefore by design constructive. Every value has exactly one type. Once the type of value is given, it can not change. This is a big difference to sets, where elements can live in any amount of different sets.

In Russell's original *ramified theory of types* [KLN04] he defined an order amongst the predicates. The lowest predicates could only reason about types. Higher predicates could reason only about lower predicates. This made the type of functions not only dependent on the types of its arguments but also on the types of predicates contained in the definition of the function. Thus, the type of a function could get very complicated even for simple functions.

In 1921 Leon Chwistek and Frank Ramsey noticed that if one would allow recursive type definitions, the complexity could be avoided. This new theory is in general referred to as *Simple Type Theory* [KLN04].

Definition 1.1: Simple Type Theory

1. 0 is a type and () is a type;
2. If $T_1, \ldots, T_n$ are simple types, then also $(T_1, \ldots, T_n)$ is a simple type.

Note that () stands for the type of all propositions and 0 is the type of all variables. $(T_1, \ldots, T_n)$ denotes the type of n-nary relations between values in T_i, for i from 1 to n. For example the predicate $\lambda a, b.a < b$ for variables a, b would be of type $(0, 0)$. For P of type (0) and Q of type (0), the predicate $\lambda a, b.P(a) \wedge Q(b)$ would be of type $((0), (0))$ [KLN04]. The theory of types has changed a lot since then.

At that time another method for dealing with Russell's paradox was invented by Ernst Zermelo: his axiomatic set theory. It was further refined by Abraham Fraenkel in 1920 to what is now known as *Zermelo-Fraenkels set theory* (ZF) [KLN04]. Mathematicians nowadays prefer using ZF over type theory, as it is more expressive.

Type theory lost most of its relevance for about 30 years. Then in the 1950s type theory finally found its use amongst computer scientists, when type checkers were added to compilers. A type checker ensures that an expression has a specific type and therefore proves that the program is well-typed. One of the first programming languages with a type checker was Fortran. Earlier type checkers existed, but only in the realm of academia.

Between 1934 and 1969 Haskell Curry and William Howard noticed that proofs could be represented as programs: A theorem can be represented by a type and the corresponding proof can be represented as a program of said type. More explicitly they noticed a relation between natural deduction and lambda calculus. This realization, now known as the *Curry-Howard-Correspondence* [KLN04], resulted in the invention of proof checking programs and automated reasoning.

2.2. The Hindley Milner Type System

One of the first systems for automated reasoning was LCF (Logic for Computable Functions) invented in 1973 by Robin Milner. To implement LCF, he invented a functional programming language called ML (Meta Language). ML uses a special system of types known as the *Hindley-Milner Type System*. This system introduces *polymorphic types*; i.e. types that can be instantiated to obtain new types. As an example, we consider the simple type theory extended by polymorphic types, as used in the Hindley-Milner type system.

Definition 2.1: Polymorphic Types for Simple Type Theory

1. Let T be a type. Then $\forall a.T$ is a polymorphic type. We call a a *type variable*.
2. Let T_0 be a type, let T be a polymorphic type. Then $T\ T_0$ is a type. Such a type is called a *type application*.
3. We call types defined in Definition 1.1 *monomorphic* types.

With this definition, the predicate $\wedge$ has type $\forall a.\forall b.(a, b)$. For the predicates P and Q of type (0), the predicate $\lambda a.\lambda b.P(a) \wedge Q(b)$ has type $\Big(\Big(\forall a.\forall b.(a, b)\Big)\ (0)\Big)\ (0)$. Note that we have derived two types for the same expression. The Hindley-Milner type system comes with two equivalence rules to ensure that every expression has a unique type. The first equivalence rule says that any bound variables may be renamed and the second that type applications can be eliminated by instantiating the polymorphic type. Therefore we have:

$$\Big(\Big(\forall a.\forall b(a, b)\Big)\ (0)\Big)\ (0) = \Big(\forall a.((0), a)\Big)\ (0) = ((0), (0))$$

Nowdays ML exists in different dialects such as SML (Standard ML), OCaml and F#.

Additionally, ML also provides an algorithm based on inference rules for inferring the type of a given expression. This algorithm provide the most general type possible. This means that it will only instantiate a polymorphic type if necessary.

2.3. Dependent Types and Refinement Types

In 1972 Per Martin-Löf introduced a new form of type theory, now known as the Martin-Löf type theory. Martin-Löf took the Curry-Howard-Correspondence and extended it such that its types are able to express statements in predicate logic. To do so, he introduced *dependent types* [KLN04].

Because dependent types have the same expressiveness as statements in predicate logic, they can be used to check proofs written in predicate logic. This checking process is still far from automatic, but it has the benefit of producing bulletproof proofs. Note that dependent types can not be automatically checked. In comparison, an extension to type theory that can be checked automatically are so-called *refinement types*. The idea behind *refinement types* is to use a predicate to restrict possible values of an existing type. Refinement types are therefore a form of *subtyping*.

The main theory behind refinement types was developed by Tim Freeman and Frank Pfenning in 1991 in the paper titled *Refinement Types for ML* [FP91]. The original

paper only allowed predicates using the logical connectives $\wedge$ and $\vee$. The underlying method for inferring the types was based on the type inference for ML.

Modern refinement types are mostly influenced by a paper in 2008 by Patrick Rondan, Ming Kawaguchi and Ranji Jhala titled *Liquid Types* [RKJ08]. Whereas the approach in 1991 used the Hindley-Milner inference with small modifications, this method introduced an additional theory on top of the Hindley-Milner inference. This new form of refinement types allow predicates written in propositional logic for integers with the order relations $\leq$ and $<$, addition and multiplication with constants. In theory one can extend the realm of allowed relations to anything that can be reasoned upon using an SMT-Solver.

2.4. An Introduction to Elm

The programming language Elm was developed by Evan Czaplicki as his master thesis [CC13] in 2012. It was initially developed as a reactive programming language. In reactive programming effects are modelled as data streams (sequences of effects). This was implemented in Elm by a data type called *Signal*. Signals allowed for "time-travel debugging": One can step forwards and backwards through the events.

While signals were a very direct implementation of the mathematical model, they turned out to be not very useful. Thus, they got replaced by a special architecture, now known as *The Elm Architecture* (TEA) [Cza21]. Nowadays, an Elm program is more like a state machine with events being state transitions:

We call the state of a program the `Model`, the events are called messages or `Msg` for short. The TEA has three exposed functions:

```
init : Model
update : Msg -> Model -> Model
view : Model -> Html Msg
```

The program start with an `init` model. The model then gets displayed on screen using the `view` function. The result of the view function is an HTML document. This resulting HTML allows the user to send messages (for example by pressing a button). These messages then get send one at the time through the `update` function. The `update` function changes the model and then calls the `view` function, resulting in an update of the HTML document. To allow impure effects like time-related actions or randomness, Elm can send and receive asynchronous messages from and to a local or remote computer.

Elm claims that it has no runtime errors [Cza21]. This is actually not completely true. There exist three sorts of runtime errors: running out of memory, non-terminating functions and comparing functions. For this thesis we can safely ignore the first sort of runtime errors, as our formal language assumes an infinite amount of memory. We can also ignore the second sort, as our formal language only allows terminating functions. As for the thrid sort of errors, the reason why Elm has problems comparing functions is that it uses the mathematical definition of equality. Two functions f and g are equal if for all x we have $f(x) = g(x)$. For infinite sets, like the real numbers, this is impossible to check numerically. Thus, Elm can not reason about $(\lambda x.x \cdot 2) = (\lambda x.x + x)$. For our thesis, we therefore do not allow comparisons between functions.

Elm has a lot of hidden features that are intended for advanced programmers. These features are mostly syntactic sugar or quality-of-life features and include recursive types (which is the only way to write non terminating functions), opaque types, extendable records and constraint type variables. For this thesis, we will not consider any of these features.

3. Formal Definition of Elm

In this chapter, we will formally define the Elm language. Section 3.1 will properly introduce types. Section 3.2 will give a definition of the Elm syntax in Backus-Naur-Form. Section 3.3 will give type inference rules to infer the proper type of an Elm program. In Section 3.4 we provide the denotational semantic of an Elm program. Section 3.5 will show that the type inference rules are sound with respect to the denotational semantic.

For this thesis we will use the following notations:

- $\mathbb{N}$ is the set of the natural numbers starting from 1.
- $\mathbb{N}_0$ is the set of the natural numbers starting from 0.
- $\mathbb{N}_a^b := \{i \in \mathbb{N}_0 \mid a \leq i \wedge i \leq b\}$ is the set of the natural numbers between a and b (including the bounds).
- We will use "." to separate a quantifier from a statement: $\forall a.F$ and $\exists a.F$, where a is a variable and F is a formula.
- Function types will be written as $a_1 \to \cdots \to a_n \to b$ instead of $a_1 \times \cdots \times a_n \to b$; thus an n-ary function is represented as a unary function whoes result is a $(n-1)$-ary function. This concept is called "currying".
- We allow the use of lambda notation for functions, i.e. $\lambda x.T$ denotes the function f defined by the equation $f(x) = T$ where T is a term.
- For a term t, we use the notation $[t]_{\{(s_1,a_1),(s_2,a_2),\ldots,(s_n,a_n)\}}$ for denoting the term-wise substitution of s_i with a_i, for $i \in \mathbb{N}_1^n$ in t. Sometimes we also write $[t]_S$ if the set $S = \{(s_1, a_1), (s_2, a_2), \ldots, (s_n, a_n)\}$ is given.
- We write $f : T_1 \nrightarrow T_2$ to say that f is a partial function from T_1 to T_2, meaning

$$\forall x \in T_1, y \in T_2.(x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2.$$

- We use $\mathcal{V}$ to denote the set of all symbols.

3.1. Defining the Hindley-Milner Type System

We will use a Hindley-Milner type system [DM82]. The main idea of such a type system is to have a defined order amongst the types. The ordering will allow us to

infer the type of any expression. In the following, we give a formal definition of this type system.

3.1.1. Notion of Types

We will first introduce types, afterwards we will define how types relate to sets by explicitly defining the values of types as finite sets. Types are split in *mono types* (monomorphic types) and *poly types* (polymorphic types). Mono types can contain so-called *type variables* that can then be bound by a quantifier within a poly type. Note that quantifiers can only occur in the outermost position, thus poly types are more general types than mono types.

Definition 1.1: Mono Types, Poly Types, Types

We define

$$
\begin{aligned}
T \text{ is a } \textit{mono type} &:\Leftrightarrow T \text{ is a type variable} \\
&\quad \vee\ T \text{ is a type application} \\
&\quad \vee\ T \text{ is an algebraic type} \\
&\quad \vee\ T \text{ is a product type} \\
&\quad \vee\ T \text{ is a function type.} \\
T \text{ is a } \textit{poly type} &:\Leftrightarrow T \text{ is of form } \forall a.T' \\
&\qquad \text{where } T' \text{ is a mono type or a poly type and } a \in \mathcal{V}. \\
T \text{ is a } \textit{type} &:\Leftrightarrow T \text{ is a mono type or a poly type.}
\end{aligned}
$$

by using the following predicates:

$$
\begin{aligned}
T \text{ is a } \textit{type variable} &:\Leftrightarrow T \in \mathcal{V} \\
T \text{ is a } \textit{type application} &:\Leftrightarrow T \text{ is of form } C\ T_1 \dots T_n \\
&\qquad \text{where } n \in \mathbb{N}, C \in \mathcal{V}, \text{ and the } T_i \text{ are mono types for} \\
&\qquad \text{all } i \in \mathbb{N}_1^n. \\
T \text{ is an } \textit{algebraic type} &:\Leftrightarrow T \text{ is of form} \\
&\qquad \mu C.C_1\ T_{1,1} \dots T_{1,k_1} \mid \dots \mid C_n\ T_{n,1} \dots T_{n,k_n} \\
&\qquad \text{such that } \exists i \in \mathbb{N}.\forall j \in \mathbb{N}_1^{k_i}.T_{i,j} \neq C \\
&\qquad \text{where } n \in \mathbb{N}, k_i \in \mathbb{N}_0 \text{ for all } i \in \mathbb{N}_1^n, C \in \mathcal{V}, \text{ and } T_{i,k_j} \\
&\qquad \text{is a mono type or } C \text{ for all } i \in \mathbb{N}_1^n \text{ and } j \in \mathbb{N}_1^{k_i}.
\end{aligned}
$$

$$
\begin{aligned}
T \text{ is a } \textit{product type} &:\Leftrightarrow T \text{ is of form } \{l_1 : T_1, \dots, l_n : T_n\} \\
&\quad \text{where } n \in \mathbb{N}_0, l_i \in \mathcal{V}, \text{ and } T_i \text{ are mono types for} \\
&\quad \text{all } i \in \mathbb{N}_1^n. \\
T \text{ is a } \textit{function type} &:\Leftrightarrow T \text{ is of form } T_1 \to T_2 \\
&\quad \text{where } T_1 \text{ and } T_2 \text{ are mono types.}
\end{aligned}
$$

We define $\mathcal{T} := \{T \mid T \text{ is a type}\}$ as the set of all types.

Note that the quantifier μC is called a *recursive quantifier*. By using the symbol C we can describe a recursive structure in a non recursive way. That said, we need to ensure that every algebraic type has a non-recursive case (called a base case). This is why we require $\exists i \in \mathbb{N}.\forall j \in \mathbb{N}_1^{k_i}.T_{i,j} \neq C$.

The presented types are in a normal form: mono types can not contain poly types and type variables are unique. We will therefore say that any term is equal to a type if it can be rewritten into one.

Definition 1.2: Type Equivalence

We say two terms T_1, T_2 are equivalent (Notation: $T_1 = T_2$) if and only if one of the following properties holds:

- $T_2 = T_1$. (Symmetry)
- T_1 is structually equivalent to T_2. (Reflexivity)
- T_1 is of form $\forall a.T_1'$ and T_2 is of form $\forall b.T_2'$ and $T_1' = [T_2']_{\{(b,a)\}}$ where T_1', T_2' are terms and $a, b \in \mathcal{V}$. (α-Conversion)
- T_1 is of form $(\forall a.T_1')\ T_3$ and $[T_1']_{\{(a,T_3)\}} = T_2$ where T_1', T_2 are terms and $a \in \mathcal{V}$. (β-Reduction)

Note that both the α-Conversion and the β-Reduction are taken from Lambda-Calculus. The underlying rewriting rules are confluent and terminating. Thus, the equivalence relation is transitive and therefore well-defined [Pie+02].

Axiom 1.1

We consider types T_i for $i \in \mathbb{N}$ in a product type as unordered, i.e.,

$$\{a : T_1, b : T_2, \dots\} = \{b : T_2, a : T_1, \dots\}$$

for all $a, b \in \mathcal{V}$ and mono types T_1, T_2.

Example 1.1

The symbol `Char` is a type variable. The expression `Sequence Char` is a type application. These expressions can be thought of as types whose implementation is unknown. The interpretation of a type variable or a type application depends on its context.

Example 1.2

$Bool = \mu\_.True \mid False$ is an algebraic type.

Note that we use the symbol _ to specify a symbol that is only used once in the definition. Multiple occurrences of _ would be seen as multiple different symbols. We call _ a *wild card*.

Example 1.3

$List = \forall a.\mu C.Empty \mid Cons\ a\ C$ is a poly type whose body $\mu C.Empty \mid Cons\ a\ C$ is an algebraic type.

Example 1.4

The empty product type $\{\}$ is a mono type. Its sometimes also called a *unit* type.

Definition 1.3: Sort, Terminal

Let $n \in \mathbb{N}$, $k_j \in \mathbb{N}$, $T_{i,j}$ be mono types, $C, C_i \in \mathcal{V}$ for all $j \in \mathbb{N}_1^n$, $i \in \mathbb{N}_1^n$ and $T = \mu C.C_1\ T_{1,1} \ldots T_{1,k_1} \mid \ldots \mid C_n\ T_{n,1} \ldots T_{n,k_n}$ be a algebraic type.

—

We call

- C_i a *terminal* of T and
- $C_i\ T_{i,1} \ldots T_{i,k_i}$ a *sort* of T for all instantiation of all type-variables in $T_{i,j}$ by mono types that do not contain type variables.

Example 1.5

The natural numbers and the integers can be defined as algebraic types using the peano axioms [Pea89]:

- 1 is a natural number.
- Every natural number has a successor.

These axioms can be used for the definition of the type.

$$Nat ::= \mu C.1 \mid Succ\ C$$

For integers, we can use the natural numbers for constructing the positive and negative numbers.

$$Int ::= \mu\_.0 \mid Pos\ Nat \mid Neg\ Nat$$

The terms $Succ\ 1$ for Nat or $Neg\ (Succ\ 1)$ for Int are sorts, whereas $Succ$ for Nat and Neg or Pos for Int are terminals. The terms 1 and 0 are both terminals and sorts.

Definition 1.4: Label

Let $n \in \mathbb{N}$, $T_i \in \mathcal{T}$, $l_i \in \mathcal{V}$ for all $i \in \mathbb{N}_1^n$.

—

We say l_i is a *label* of the product type $\{l_1 : T_1, .., l_n : T_n\}$ for all $i \in \mathbb{N}_1^n$.

We define

$$T_1 \times \cdots \times T_n := \{1 : T_1, \ldots, n : T_n\}$$

as the *ordered product type* with n components.

The most basic example of a product type is a record. Tuples can be represented as ordered product types.

Definition 1.5: Bound, Free, Set of Free Variables

Let $a \in \mathcal{V}$ and $T \in \mathcal{T}$.

—

We say

- a is *free* in T $:\Leftrightarrow a \in \text{free}(T)$
- a is *bound* in T $:\Leftrightarrow a \notin \text{free}(T)$ and a occurs in T.

where

$$
\begin{aligned}
\text{free} : \mathcal{T} &\to \mathcal{P}(\mathcal{V}) \\
\text{free}(a) :=& \{a\} \\
\text{free}(C\ T_1 \dots T_n) :=& \bigcup_{i \in \mathbb{N}_1^n} \text{free}(T_i) \\
\text{free}\begin{pmatrix} \mu C. \\ C_1\ T_{1,1} \dots\ T_{1,k(1)} \\ | \dots \\ |\ C_n\ T_{n,1} \dots\ T_{n,k(n)} \end{pmatrix} :=& \bigcup_{i \in \mathbb{N}_0^n} \bigcup_{j \in \mathbb{N}_0^{k_i}} \begin{cases} \varnothing & \text{if } T_{i,j} = C \\ \text{free}(T_{i,j}) & \text{else} \end{cases} \\
\text{free}(\{\_ : T_1, \dots, \_ : T_n\}) :=& \bigcup_{i \in \mathbb{N}_1^n} \text{free}(T_i) \\
\text{free}(T_1 \to T_2) :=& \text{free}(T_1) \cup \text{free}(T_2) \\
\text{free}(\forall a.T) :=& \text{free}(T) \backslash \{a\}
\end{aligned}
$$

A poly type can be instantiated with a mono type by applying β-Reduction. The result will again be a type.

Definition 1.6: Type Instatiation

We define the following:

$$
\begin{aligned}
\text{Inst} : \mathcal{T} &\to (\mathcal{V} \nrightarrow \mathcal{T}) \to \mathcal{T} \\
\text{Inst}(T, \Theta) :=& \begin{cases} \text{Inst}(T_1, \Theta) & \text{If } T \text{ is of form } \forall a.T_1 \text{ and } (a, \_) \in \Theta \\ & \text{where } a \in \mathcal{V} \text{ and } T_1 \in \mathcal{T} \\ \forall a.\text{Inst}(T_1, \Theta) & \text{If } T \text{ is of form } \forall a.T_1 \text{ and } (a, \_) \notin \Theta \\ & \text{where } a \in \mathcal{V} \text{ and } T_1 \in \mathcal{T} \\ [T]_\Theta & \text{else} \end{cases}
\end{aligned}
$$

The type instatiation gives raise to a partial order $\sqsubseteq$:

Definition 1.7: Type Order

Let $n, m \in \mathbb{N}_0$, $T_1, T_2 \in \mathcal{T}$, a_i for all $i \in \mathbb{N}^n$ and $b_j \in \mathcal{V}$ for all $j \in \mathbb{N}_0^m$.

—

We define the partial order $\sqsubseteq \subseteq \mathcal{T} \times \mathcal{T}$ such that $\forall a_1 \dots \forall a_n.T_1 \sqsubseteq \forall b_1 \dots \forall b_m.T_2$ if and only if there exists a $\Theta = \{(a_i, T_i') \mid i \in \mathbb{N}_1^n, T_i' \in \mathcal{T}\}$ such that $T_2 = \text{Inst}(T_1, \Theta)$ and $b_j \notin \text{free}(\forall a_1 \dots \forall a_n.T_1)$ for all $j \in \mathbb{N}_0^m$.

Example 1.6

$\forall a.a$ is the smallest type in the type system. The partial order forms a tree structure with $\forall a.a$ at the root and different branches for $\forall a.C\ a$, $\forall a.\forall b.a \to b$ and so on. The mono types form the leaves of the tree.

3.1.2. Interpretation of Types

Before we interpret a type, we will first introduce a set of labelled elements as a record.

Definition 1.8: Record

Let n in $\mathbb{N}$, l_i be symbols, t_i terms for all i in $\mathbb{N}_1^n$.

—

We define

$$\{l_1 = t_1, \ldots, l_n = t_n\} : \{l_1, \ldots, l_n\} \to \{t_1, \ldots, t_n\}$$
$$\{l_1 = t_1, \ldots, l_n = t_n\}(l) := t_i \text{ such that } l = l_i \text{ for some } i \in \mathbb{N}_1^n.$$

Note that values of an ordered product type are equivalent to values of a tuple:

$$\forall i \in \mathbb{N}_1^n.\{1 = t_1, \ldots, n = t_n\}(i) = t_i$$

Thus, we will use the notation of tuples for values of an ordered product type.

Definition 1.9: Application Constructor

Let $n \in \mathbb{N}_0$. Let T be a mono type. Let $\{a_1, \ldots, a_n\} = \text{free}(T)$ where $a_i \in \mathcal{V}$ for all $i \in \mathbb{N}_1^n$.

—

We call the function

$$\overline{\forall a_1 \ldots a_n.T} : \underbrace{\mathcal{T} \to \cdots \to \mathcal{T}}_{n \text{ times}} \to \mathcal{T}$$
$$(\overline{\forall a_1 \ldots a_n.T})(T_1, \ldots, T_n) := \text{Inst}(\forall a_1 \ldots a_n.T, \{(a_i, T_i) \mid i \in \mathbb{N}_0^n\})$$

the *application constructor* of T.

Therefore, for a given type T', the application constructor of T' is notated as $\overline{T'}$.

Note that mono types with no free variables are considered to be application constructors with no arguments.

Definition 1.10: Type Context

$\Gamma : \mathcal{V} \nrightarrow \mathcal{T}$ is called a *type context*.

Definition 1.11: Values

Let $\mathcal{S}$ be the class of all finite sets and Γ be a type context.

—

We define

$$\begin{aligned}
\text{values}_\Gamma &: \mathcal{V} \to \mathcal{S} \\
\text{values}_\Gamma(a) &:= \text{values}_\Gamma(\Gamma(a)) \\
\text{values}_\Gamma(C\ T_1\ \ldots\ T_n) &:= \text{values}_\Gamma(\overline{\Gamma(C)}(T_1, \ldots, T_n)) \\
\text{values}_\Gamma \begin{pmatrix} \mu C. \\ \mid C_1\ T_{1,1} \ldots\ T_{1,k_1} \\ \mid \ldots \\ \mid C_n\ T_{n,1} \ldots\ T_{n,k_n} \end{pmatrix} &:= \bigcup_{i \in \mathbb{N}_0} \text{rvalues}_\Gamma \begin{pmatrix} & \mu C. \\ i, & \mid C_1\ T_{1,1} \ldots\ T_{1,k_1} \\ & \mid \ldots \\ & \mid C_n\ T_{n,1} \ldots\ T_{n,k_n} \end{pmatrix} \\
\text{values}_\Gamma(\{l_1 : T_1, \ldots, l_n : T_n\}) &:= \Big\{ \{l_1 = t_1, \ldots, l_n = t_n\} \\
&\qquad \mid \forall i \in \mathbb{N}_1^n . t_i \in \text{values}_\Gamma(T_i) \Big\} \\
\text{values}_\Gamma(T_1 \to T_2) &:= \{ f \mid f : \text{values}_\Gamma(T_1) \to \text{values}_\Gamma(T_2) \} \\
\text{values}_\Gamma(\forall a.T) &:= \lambda b.\text{values}_{\{(a,b)\} \cup \Gamma}(T) \text{ where the symbol } b \text{ does} \\
&\qquad \text{not occur in } T.
\end{aligned}$$

using the following helper function.

Let $l \in \mathbb{N}, T := \mu C.\ C_1\ T_{1,1} \ldots\ T_{1,k_1} \mid \ldots \mid C_n\ T_{n,1} \ldots\ T_{n,k_n}$ in

$$\begin{aligned}
\text{rvalues}_\Gamma(0, T) &:= \left\{ C_i\ v_{i,1} \ldots v_{i,n} \;\middle|\; \begin{array}{l} i \in \mathbb{N}_1^n \\ \wedge \forall j \in \mathbb{N}_1^{k_i} . T_{i,j} \neq C \wedge v_{i,j} \in \text{values}_\Gamma(T_{i,j}) \end{array} \right\} \\
\text{rvalues}_\Gamma(l+1, T) &:= \left\{ C_i\ v_{i,1} \ldots v_{i,n} \;\middle|\; \begin{array}{l} i \in \mathbb{N}_1^n \\ \wedge \forall j \in \mathbb{N}_1^{k_i} . v_j \in \begin{cases} \text{rvalues}_\Gamma(l, T) & \text{if } T_{i,j} = C \\ \text{values}_\Gamma(T_{i,j}) & \text{else} \end{cases} \end{array} \right\}
\end{aligned}$$

The base case of this recursive function is in $\text{rvalues}_\Gamma(0, T)$ for a given T.

As an example we will prove that the values of *Nat* from Example 1.5 are isomorphic to the natural numbers.

To simplify the theorem we will introduce a new notation: For any $n \in \mathbb{N}_0$ we define $Succ^n\ 1 := \underbrace{Succ\ \ldots\ Succ}_{n \text{ times}}\ 1$. Note that $Succ^0\ 1 = 1$.

Theorem 1.1

Let the algebraic type Nat be defined as $Nat := \mu C.1|Succ\ C$. Let $<_{\mathbb{N}}: \mathbb{N} \times \mathbb{N}$ be the well-order such that $a <_{\mathbb{N}} b :\Leftrightarrow \exists c \in \mathbb{N}.a = b + c$. Let $<_{Nat}: Nat \times Nat$ be a order such that $Succ^a\ 1 <_{Nat} Succ^b\ 1 :\Leftrightarrow a + 1 <_{\mathbb{N}} b + 1$.

—

Then we have:

$$(\text{values}(Nat), <_{Nat}) \cong (\mathbb{N}, <_{\mathbb{N}})$$

Proof. We show by induction over $n \in \mathbb{N}_0$ that

$$\text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C) = \{Succ^i\ 1 \mid i \in \mathbb{N}_0^n\}. \tag{3.1}$$

Base case: $\text{rvalues}_\Gamma(0, \mu C.1 \mid Succ\ C) = \{1\} = \{Succ^0\}$. This is true.
Inductive step:
Assuming $\text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C) = \{Succ^i\ 1 \mid i \in \mathbb{N}_0^n\}$, we will prove $\text{rvalues}_\Gamma(n+1, \mu C.1 \mid Succ\ C) = \{Succ^i\ 1 \mid i \in \mathbb{N}_0^{n+1}\}$.

$$\begin{aligned}
&\text{rvalues}_\Gamma(n+1, \mu C.1|Succ\ C) \\
&= \left\{ C_i\ v_{i,1} \ldots v_{i,n} \;\middle|\; \begin{array}{l} i \in \mathbb{N}_1^n \wedge C_1 = 1 \wedge C_2 = Succ \wedge k_1 = 0 \wedge k_2 = 1 \wedge T_{2,1} = C \\ \wedge\ \forall j \in \mathbb{N}_1^{k(i)}.v_{i,j} \in \begin{cases} \text{rvalues}_\Gamma(n, T) & \text{if } T_{i,j} = C \\ \text{values}_\Gamma(T_{i,j}) & \text{else} \end{cases} \end{array} \right\} \\
&= \left\{ C_i\ v_{i,1} \ldots v_{i,n} \;\middle|\; \begin{array}{l} i \in \mathbb{N}_1^n \wedge C_1 = 1 \wedge C_2 = Succ \wedge k_1 = 0 \wedge k_2 = 1 \wedge T_{2,1} = C \\ \wedge \forall j \in \mathbb{N}_1^{k(i)}.v_{i,j} \in \begin{cases} \{Succ^k\ 1 \mid k \in \mathbb{N}_0^n\} & \text{if } T_{i,j} = C \\ \text{values}_\Gamma(T_{i,j}) & \text{else} \end{cases} \end{array} \right\} \\
&= \{1\} \cup \{Succ\ v | v \in \{Succ^i\ 1 \mid i \in \mathbb{N}_0^n\}\} \\
&= \{Succ^i\ 1 \mid i \in \mathbb{N}_0^{n+1}\}
\end{aligned}$$

Now we will prove

$$\text{values}(\mu C.1 \mid Succ\ C) = \{Succ^n\ 1 \mid n \in \mathbb{N}_0\}. \tag{3.2}$$

"$\subseteq$": Let $x \in \text{values}(\mu C.1 \mid Succ\ C)$. We show

$$x \in \{Succ^n\ 1 \mid n \in \mathbb{N}_0\}.$$

We know

$$\text{values}(\mu C.1 \mid Succ\ C) = \bigcup_{i \in \mathbb{N}_0} \text{rvalues}_\Gamma(i, \mu C.1 \mid Succ\ C)$$

and

$$\text{rvalues}_\Gamma(i, \mu C.1 \mid Succ\ C) \overset{(3.1)}{=} \{Succ^k\ 1 \mid k \in \mathbb{N}_0^i\}.$$

This means, there exists an $i \in \mathbb{N}_0$ such that

$$x \in \{Succ^k\ 1 \mid k \in \mathbb{N}_0^i\}.$$

Therefore there exists a $k \in \mathbb{N}_0^i$, such that

$$x = Succ^k\ 1.$$

Thus, in conclusion,

$$x \in \{Succ^n\ 1 \mid n \in \mathbb{N}_0\}.$$

"$\supseteq$": Let $x \in \{Succ^n\ 1 \mid n \in \mathbb{N}_0\}$. We show

$$x \in \text{values}(\mu C.1 | Succ\ C).$$

We know

$$\text{values}(\mu C.1 \mid Succ\ C) = \bigcup_{n \in \mathbb{N}_0} \text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C).$$

Thus, it is suffice to show

$$x \in \bigcup_{n \in \mathbb{N}_0} \text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C).$$

From $x \in \{Succ^n\ 1 \mid n \in \mathbb{N}_0\}$ we know that there exists a $n \in \mathbb{N}_0$ such that

$$x = Succ^n\ 1.$$

Using said n, we now construct $\{Succ^i\ 1 \mid i \in \mathbb{N}_0^n\}$. We know

$$\begin{aligned}\{Succ^i\ 1 \mid i \in \mathbb{N}_0^n\} &\overset{(3.1)}{=} \text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C) \\ &\subseteq \bigcup_{n \in \mathbb{N}_0} \text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C).\end{aligned}$$

As $x \in \{Succ^i\ 1 \mid i \in \mathbb{N}_0^n\}$ and $\{Succ^i\ 1 \mid i \in \mathbb{N}_0^n\} \subseteq \bigcup_{n \in \mathbb{N}_0} \text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C)$ we conclude

$$x \in \bigcup_{n \in \mathbb{N}_0} \text{rvalues}_\Gamma(n, \mu C.1 \mid Succ\ C).$$

To summarize, we have just shown that

$$\text{values}(Nat) = \text{values}(\mu C.1 \mid Succ\ C) \overset{(3.2)}{=} \{Succ^n\ 1 \mid n \in \mathbb{N}_0\}.$$

For the last step, we define a bijection.

$$\begin{aligned} &h : \{Succ^n\ 1 \mid n \in \mathbb{N}_0\} \to \mathbb{N} \\ &h(Succ^n\ 1) = n + 1 \\ &h^{-1}(n) = Succ^{n-1}\ 1 \end{aligned}$$

Thus

$$|\{Succ^n\ 1 \mid n \in \mathbb{N}_0\}| = |\mathbb{N}|.$$

For all $n, m \in \mathbb{N}_0$ we see that

$$\begin{aligned} Succ^n\ 1 <_{Nat} Succ^m\ 1 &\Leftrightarrow n + 1 <_{\mathbb{N}} m + 1 \\ &\Leftrightarrow h(Succ^n\ 1) <_{\mathbb{N}} h(Succ^m\ 1). \end{aligned}$$

And therefore h is a isomorphism, thus

$$(\text{values}(Nat), <_{Nat}) \cong (\mathbb{N}, <_{\mathbb{N}}).$$

□

3.2. Syntax

Elm differentiates variables depending on the capitalization of the first letter. For the formal language we define `<upper-var>` for variables with the first letter capitalized and `<lower-var>` for variables without.

Syntactically, we can build our types from booleans, integers, lists, tuples, records, functions, custom types and type variables.

We will define our syntax in a Backus-Naur-Form [Bac59].

Definition 2.1: Type Signiture Syntax

Given two variable domains `<upper-var>` and `<lower-var>`, we define the following syntax:

```
<list-type-fields> ::= ""
                     | <lower-var> ":" <type> "," <list-type-fields>

<list-type> ::= "" | <type> <list-type>

<type> ::="Bool"
        | "Int"
        | "List" <type>
        | "(" <type> "," <type> ")"
        | "{" <list-type-fields> "}"
        | <type> "->" <type>
        | <upper-var> <list-type>
        | <lower-var>
```

Because Elm is a pure functional programming language, a program is just a single expression.

Definition 2.2: Expression Syntax

Given two variable domains `<upper-var>` and `<lower-var>`, we define the following syntax:

```
<list-exp-field> ::= <lower-var> "=" <exp>
                   | <lower-var> "=" <exp> "," <list-exp-field>

<maybe-exp-sign> ::= "" | <lower-var> ":" <type> ";"

<bool> ::= "True" | "False"

<int> ::= "0" | "-1" | "1" | "-2" | "2" | ...

<list-exp> ::= "" | <exp> "," <list-exp>
```

```
<exp> ::= "foldl"
        | "(::)"
        | "(+)" | "(-)" | "(*)" | "(//)"
        | "(<)" | "(==)"
        | "not" | "(&&)" | "(||)"
        | "if" <exp> "then" <exp> "else" <exp>
        | "{" <list-exp-field> "}"
        | "{}"
        | "{" <lower-var> "|" <list-exp-field> "}"
        | <lower-var> "." <lower-var>
        | "let" <maybe-exp-sign> <lower-var> "=" <exp> "in" <exp>
        | <exp> <exp>
        | <bool>
        | <int>
        | "[" <list-exp> "]"
        | "(" <exp> "," <exp> ")"
        | "\" <lower-var> "->" <exp>
        | <upper-var>
        | <lower-var>
```

Additionally, Elm also allows global constants, type aliases and custom types.

Definition 2.3: Statement Syntax

Given two variable domains `<upper-var>` and `<lower-var>`, we define the following syntax:

```
<list-statement-var> ::= "" | <lower-var> <list-statement-var>

<list-statement> ::= "" | <statement> ";" <list-statement>

<maybe-statement-sign> ::= "" | <lower-var> ":" <type> ";"

<statement> ::= <maybe-statement-sign> <lower-var> "=" <exp>
              | "type" "alias" <upper-var> <list-statement-var>
                "=" <type>

<maybe-main-sign> ::= "" | "main" ":" <type> ";"
```

```
<program> ::= <list-statement> <maybe-main-sign> "main" "=" <exp>
```

Example 2.7

Using this syntax we can now write a function that reverses a list.

```
reverse : List a -> List a;
reverse =
  foldl (::) [];

main : Int;
main =
  case reverse [1,2,3]  of
  [
    a :: _ ->
      a;
    _ ->
      -1
  ]
```

`foldl` iterates over the list from left to right. It takes the function `(::)`, that appends an element to a list, and the empty list as the starting list. The `main` function reverses the list and returns the first element: 3. Elm requires you also provide return values for other cases that may occur, like the empty list. In that case we just return -1. This will never happened, as long as the reverse function is correctly implemented.

3.3. Type Inference

Now that we have defined a syntax and a type system for our language, we want to introduce rules how to obtain the type of a given program written in our language.

3.3.1. Typing Judgments

A type system is a set of inference rules to derive various kinds of typing judgments. These *inference rules* have the following form

$$\frac{P_1 \dots P_n}{C}$$

where the judgments P_1 up to P_n are the premises of the rule and the judgment C is its conclusion.

We can read it in two ways:

- "If all premises hold then the conclusion holds as well" or
- "To prove the conclusion we need to prove all premises".

We will now provide a judgment for every production rule defined in Section 3.2. Ultimately, we will have a judgment $p : T$ which indicates that a program p is of a type T and therefore well-formed.

If the type T is known then we talk about *type checking* else we call the process of finding the judgment *type inference*.

Type Signature Judgments

For type signature judgments, let Γ be a type context, $T \in \mathcal{T}$ and $a_i \in \mathcal{V}, T_i \in \mathcal{T}$ for all $i \in \mathbb{N}_1^n$ and $n \in \mathbb{N}$.

For *ltf* $\in$ `<list-type-fields>` the judgment has the form

$$\Gamma \vdash \mathit{ltf} : \{a_1 : T_1, \ldots, a_n : T_n\}$$

which can be read as "given Γ, ltf has the type $\{a_1 : T_1, \ldots, a_n : T_n\}$".

For $lt \in$ `<list-type>` the judgment has the form

$$\Gamma \vdash \mathit{lt} : (T_1, \ldots, T_n)$$

which can be read as "given Γ, lt defines the list $(T_1, \ldots, T_n)$".

For $t \in$ `<type>` the judgment has the form

$$\Gamma \vdash t : T$$

which can be read as "given Γ, t has the type T".

Expression Judgments

For expression judgments, let Γ, Δ be type contexts, $T \in \mathcal{T}$, $a \in \mathcal{V}$ and $T_i \in \mathcal{T}, a_i \in \mathcal{V}$ for all $i \in \mathbb{N}_0^n, n \in \mathbb{N}$.

For *lef* $\in$ `<list-exp-field>` the judgment has the form

$$\Gamma, \Delta \vdash \mathit{lef} : \{a_1 : T_1, \ldots, a_n : T_n\}$$

which can be read as "given Γ and Δ, *lef* has the type $\{a_1 : T_1, \ldots, a_n : T_n\}$".

For $mes \in$ `<maybe-exp-sign>` the judgment has the form

$$\Gamma, mes \vdash a : T$$

which can be read as "given Γ, a has the type T under the assumption mes".

For $b \in$ `<bool>` the judgment has the form

$$b : T$$

which can be read as "b has the type T".

For $i \in$ `<int>` the judgment has the form

$$e : T$$

which can be read as "i has the type T".

For $le \in$ `<list-exp>` the judgment has the form

$$\Gamma, \Delta \vdash le : List\ T$$

which can be read as "given Γ and Δ, le has the type $List\ T$".

For $e \in$ `<exp>` the judgment has the form

$$\Gamma, \Delta \vdash e : T$$

which can be read as "given Γ and Δ, e is of type T".

Statement Judgments

For statement judgments, let $\Gamma, \Gamma_1, \Gamma_2, \Delta, \Delta_1, \Delta_2$ be a type contexts, $T, T_1, T_2 \in \mathcal{T}$, $a \in \mathcal{V}$ and $T_i, A_i \in \mathcal{T}, a_i \in \mathcal{V}$ for $i \in \mathbb{N}_0^n$ and $T_{i,j} \in \mathcal{T}$ for $i \in \mathbb{N}_0^n, n \in \mathbb{N}, j \in \mathbb{N}_0^{k_i}$ and $k_i \in \mathbb{N}$.

For $lsv \in$ `<list-statement-var>` the judgment has the form

$$lsv : (a_1, \ldots, a_n)$$

which can be read as "lsv describes the list $(a_1, \ldots, a_n)$".

For $ls \in$ `<list-statement>` the judgment has the form

$$\Gamma_1, \Delta_2, ls \vdash \Gamma_2, \Delta_2$$

which can be read as "the list of statements ls maps Γ_1 to Γ_2 and Δ_1 to Δ_2".

For $mss \in$ `<maybe-statement-sign>` the judgment has the form

$$\Gamma, mss \vdash a : T$$

which can be read as "given Γ, a has the type T_2 under the assumption mss".

For $s \in$ `<statement>` the judgment has the form

$$\Gamma_1, \Delta_1, s \vdash \Gamma_2, \Delta_2$$

which can be read as "the statement s maps Γ_1 to Γ_2 and Δ_1 to Δ_2".

For $mms \in$ `<maybe-main-sign>` the judgment has the form

$$\Gamma, mms \vdash \text{main} : T$$

which can be read as "the main function has type T under the assumtion mms".

For $prog \in$ `<program>` the judgment has the form

$$prog : T$$

which can be read as "the program $prog$ is well-formed and has the type T".

3.3.2. Auxiliary Definitions

We will assume that "T is a mono type" and "T is a type variable" is definied. $T_1 = T_2$ denotes the equiality of two given types T_1 and T_2.

We will write $\{a_1, \ldots, a_n\} = \text{free}(T)$ to denote all free variables $a_1, \ldots, a_n$ of T.

Instantiation, Generalization

The type system that we are using is polymorphic, meaning that whenever a judgment holds for a type, it will also hold for any type that is more specific. To counter this we will force the types in a judgment to be unique by explicitly stating whenever we want to use a more specific or general type.

Definition 3.1: Most General Type

Let Γ be a type context and $T \in \mathcal{T}$.

—

We define $\overline{\Gamma} : \Gamma \to \mathcal{T}$ as

$$\begin{aligned}\overline{\Gamma}(T) :=& \forall a_1 \ldots \forall a_n . T_0 \\ & \text{such that } \{a_1, \ldots, a_n\} = \text{free}(T_0) \setminus \{a \mid (a, \_) \in \Gamma\} \\ & \text{where } a_i \in \mathcal{V} \text{ for } i \in \mathbb{N}_0^n \text{ and } T_0 \text{ is the mono type of } T.\end{aligned}$$

We say $\overline{\Gamma}(T)$ is *the most general type* of T.

The most general type ensures that all type variables are bound by either an quantifier or a type alias in the type context Γ. It also ensure that every type variable bound by a quantifier occures in the mono type T_0.

The act of replacing types with more general onces, by binding free variables, is called *Generalization* and the act of replacing are more general type with a more specific type is called *Instantiation*. Both rules are typically in the text books [Pie04] introduced as an additional inference rule.

Predefined Types

Additionally, we define

$$\begin{aligned}\mathit{Bool} &:= \mu\_.\mathit{True} \mid \mathit{False} \\ \mathit{Nat} &:= \mu C.1 \mid \mathit{Succ}\ C \\ \mathit{Int} &:= \mu\_.0 \mid \mathit{Pos}\ \mathit{Nat} \mid \mathit{Neg}\ \mathit{Nat} \\ \mathit{List} &:= \forall a.\mu C.[\ ] \mid \mathit{Cons}\ a\ C.\end{aligned}$$

3.3.3. Inference Rules for Type Signatures

We will now describe the inference rules for type signatures. This is nothing more than a translation from `<type>` to $\mathcal{T}$.

Inference Rules for `<list-type-fields>`

Judgment: $\Gamma \vdash \mathit{ltf} : \{a_1 : T_1, \ldots, a_n : T_n\}$

$$\Gamma \vdash \texttt{""} : \{\}$$

$$\frac{\Gamma \vdash t : T_0 \quad \Gamma \vdash \mathit{ltf} : \{a_1 : T_1, \ldots, a_n : T_n\} \quad \{a_0 : T_0, a_1 : T_1, \ldots, a_n : T_n\} = T}{\Gamma \vdash a_0\ \texttt{":"}\ t\ \texttt{","}\ \mathit{ltf} : T}$$

The type context Γ is used for the judgment $\Gamma \vdash t : T_0$ that turns the type signature t into a type T_0.

Inference Rules for `<list-type>`

Judgment: $\Gamma \vdash lt : (T_1, \ldots, T_n)$

$$\Gamma \vdash \texttt{""} : ()$$

$$\frac{\Gamma \vdash t : T_0 \quad \Gamma \vdash lt : (T_1, \ldots, T_n) \quad (T_0, T_1, \ldots, T_n) = T}{\Gamma \vdash t\ lt : T}$$

Inference Rules for `<type>`

Judgment: $\Gamma \vdash t : T$

$$\frac{Bool = T}{\Gamma \vdash \texttt{"Bool"} : T}$$

$$\frac{Int = T}{\Gamma \vdash \texttt{"Int"} : T}$$

$$\frac{List\ T_2 = T_1 \quad \Gamma \vdash t : T_2}{\Gamma \vdash \texttt{"List"}\ \texttt{t} : T_1}$$

$$\frac{(T_1, T_2) = T_0 \quad \Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \texttt{"("}\ t_1\ \texttt{","}\ t_2\ \texttt{")"} : T_0}$$

$$\frac{\Gamma \vdash ltf : T}{\Gamma \vdash \texttt{"\{"}\ ltf\ \texttt{"\}"} : T}$$

$$\frac{T_1 \to T_2 = T_0 \quad \Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \to t_2 : T_0}$$

$$\frac{(c, T') \in \Gamma \quad \Gamma \vdash l : (T_1, \ldots, T_n) \quad \overline{T'}\ T_1 \ldots T_n = T}{\Gamma \vdash c\ l : T}$$

For a given type T we write the application constructor as $\overline{T}$.

$$\frac{\forall a.a = T}{\Gamma \vdash a : T}$$

3.3.4. Inference Rules for Expressions

We will now go over the inference rules for expressions. In Elm, any expression has a type with respect to a given type context Γ and variable context Δ.

Inference Rules for `<list-exp-field>`

Judgment: $\Gamma, \Delta \vdash lef : \{a_1 : T_1, \ldots, a_n : T_n\}$

$$\frac{\Gamma, \Delta \vdash e : T}{\Gamma, \Delta \vdash a \texttt{ "=" } e : \{a : T\}}$$

$$\frac{\Gamma, \Delta \vdash lef : T \quad \Gamma, \Delta \vdash e : T_0 \quad \{a_0 : T_0, \ldots, a_n : T_n\} = T}{\Gamma, \Delta \vdash a_0 \texttt{ "=" } e \texttt{ "," } lef : T}$$

Inference Rules for `<maybe-exp-sign>`

Judgment: $\Gamma, mes \vdash a : T$

$$\Gamma, \texttt{""} \vdash a : T$$

If no argument is given, then we do nothing.

$$\frac{\Gamma \vdash t : T \quad a_1 = a_2}{\Gamma, a_1 \texttt{ ":" } t \texttt{ ";" } \vdash a_2 : T}$$

If we have a variable a_1 and a type T, then the variables a_2 need to match. The type signature t defines the type of a_2.

Inference Rules for `<bool>`

Judgment: $b : T$

$$b : Bool$$

Inference Rules for `<int>`

Judgment: $i : T$

$$i : Int$$

We have proven in Theorem 1.5 that *Nat* is isomorph to $\mathbb{N}$. Is should be trivial to therefore conclude that *Int* is isomorph to $\mathbb{Z}$. And therefore this rule is justified.

Inference Rules for `<list-exp>`

Judgment: $\Gamma, \Delta \vdash le : List\ T$

$$\Gamma, \Delta \vdash \texttt{""} : \forall a.List\ a$$

$$\frac{\Gamma, \Delta \vdash e : T \quad \Gamma, \Delta \vdash le : List\ T}{\Gamma, \Delta \vdash e\ \texttt{","}\ le : List\ T}$$

Inference Rules for `<exp>`

Judgment: $\Gamma, \Delta \vdash e : T$

$$\Gamma, \Delta \vdash \texttt{"foldl"} : \forall a.\forall b.(a \to b \to b) \to b \to List\ a \to b$$

$$\Gamma, \Delta \vdash \texttt{"(::)"} : \forall a.a \to List\ a \to List\ a$$

$$\Gamma, \Delta \vdash \texttt{"(+)"} : Int \to Int \to Int$$

$$\Gamma, \Delta \vdash \texttt{"(-)"} : Int \to Int \to Int$$

$$\Gamma, \Delta \vdash \texttt{"(*)"} : Int \to Int \to Int$$

$$\Gamma, \Delta \vdash \texttt{"(//)"} : Int \to Int \to Int$$

$$\Gamma, \Delta \vdash \texttt{"(<)"} : Int \to Int \to Bool$$

$$\Gamma, \Delta \vdash \texttt{"(==)"} : Int \to Int \to Bool$$

$$\Gamma, \Delta \vdash \texttt{"not"} : \mathit{Bool} \to \mathit{Bool}$$

$$\Gamma, \Delta \vdash \texttt{"(\&\&)"} : \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$$

$$\Gamma, \Delta \vdash \texttt{"(||)"} : \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$$

$$\frac{\Gamma, \Delta \vdash e_1 : \mathit{Bool} \quad \Gamma, \Delta \vdash e_2 : T \quad \Gamma, \Delta \vdash e_3 : T}{\Gamma, \Delta \vdash \texttt{"if"}\ e_1\ \texttt{"then"}\ e_2\ \texttt{"else"}\ e_3 : T}$$

$$\frac{\Gamma, \Delta \vdash \mathit{lef} : \{a_1 : T_1, \ldots, a_n : T_n\}}{\Gamma, \Delta \vdash \texttt{"\{"}\ \mathit{lef}\ \texttt{"\}"} : \{a_1 : T_1, \ldots, a_n : T_n\}}$$

$$\Gamma, \Delta \vdash \texttt{"\{\}"} : \{\}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash \mathit{lef} : \{a_1 : T_1, \ldots, a_n : T_n\} \\ \Gamma, \Delta \vdash (a, \overline{\Gamma}(T_0)) \in \Delta \quad T_0 = \{a_1 : T_1, \ldots, a_n : T_n, \ldots\}\end{array}}{\Gamma, \Delta \vdash \texttt{"\{"}\ a\ \texttt{"|"}\ \mathit{lef}\ \texttt{"\}"} : T_0}$$

$$\frac{(a_1, \{a_2 : T, \ldots\}) \in \Delta}{\Gamma, \Delta \vdash a_1\texttt{"."}a_2 : T}$$

$$\frac{\begin{array}{c}(a, \_) \notin \Delta \quad \Gamma, \Delta \vdash e_1 : T_1 \quad \mathit{mes} : T_1 \vdash a : T_1 \\ \Gamma, \Delta \cup \{(a, \overline{\Gamma}(T_1))\} \vdash e_2 : T_2\end{array}}{\Gamma, \Delta \vdash \texttt{"let"}\ \mathit{mes}\ a\ \texttt{"="}\ e_1\ \texttt{"in"}\ e_2 : T_2}$$

$$\frac{\Gamma, \Delta \vdash e_1 : T_1 \to T_2 \quad \Gamma, \Delta \vdash e_2 : T_1}{\Gamma, \Delta \vdash e_1\ e_2 : T_2}$$

$$\frac{b : T}{\Gamma, \Delta \vdash b : T}$$

$$\frac{i : T}{\Gamma, \Delta \vdash i : T}$$

$$\frac{\Gamma, \Delta \vdash le : T}{\Gamma, \Delta \vdash \texttt{"["}\ le\ \texttt{"]"} : T}$$

$$\frac{\Gamma, \Delta \vdash e_1 : T_1 \quad \Gamma, \Delta \vdash e_2 : T_2}{\Gamma, \Delta \vdash \texttt{"("}\ e_1\ \texttt{","}\ e_2\texttt{")"} : (T_1, T_2)}$$

$$\frac{\Gamma, \Delta \cup \{(a, \overline{\Gamma}(T_1))\} \vdash e : T_2}{\Gamma, \Delta \vdash \texttt{"\textbackslash"}\ a\ \texttt{"->"}\ e : T_1 \to T_2}$$

$$\frac{(c, \overline{\Gamma}(T)) \in \Delta}{\Gamma, \Delta \vdash c : T}$$

$$\frac{(a, \overline{\Gamma}(T)) \in \Delta}{\Gamma, \Delta \vdash a : T}$$

Example 3.8

In Example 2.7 we have looked at the syntax for a list reversing function. We can now check the type $T_0 = \forall a.\mathit{List}\ a \to \mathit{List}\ a$ of the `reverse` function for $\Gamma = \Delta = \varnothing$, $\Delta = \varnothing$. The body of the *reverse* function is as follows:

```
foldl (::) []
```

For deriving the judgment the following sets are required.

$$\dfrac{\dfrac{\dfrac{}{\varnothing, \varnothing \vdash \texttt{"foldl"} : T_2} \quad \dfrac{}{\varnothing, \varnothing \vdash \texttt{"(::)"} : \forall a.\mathit{List}\ a \to \mathit{List}\ a}}{\varnothing, \varnothing \vdash \texttt{"foldl (::)"} : T_1} \quad \dfrac{\dfrac{}{\varnothing, \varnothing \vdash \texttt{""} : \forall a.a}}{\varnothing, \varnothing \vdash \texttt{"[]"} : \forall a.\mathit{List}\ a}}{\varnothing, \varnothing \vdash \texttt{"foldl (::) []"} : T_0}$$

where $T_1 = \forall a.\mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a$ and $T_2 = \forall a.(\mathit{List}\ a \to \mathit{List}\ a) \to \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a$.

3.3.5. Inference Rules for Statements

We now provide the inference rules for statements. We can model statements as functions and a list of statements as a composition of functions.

Inference Rules for `<list-statement-var>`

Judgment: $\mathit{lsv} : (a_1, \ldots, a_n)$

$$\texttt{""} : ()$$

$$\frac{lsv : (a_1, \ldots, a_n)}{a_0 \ lsv : (a_0, a_1, \ldots, a_n)}$$

Inference Rules for `<list-statement-sort>`

Judgment: $lss : (c_1 : (T_{1,1}, \ldots, T_{1,k_1}), \ldots, c_n : (T_{n,1}, \ldots, T_{n,k_n}))$

$$\frac{\Gamma \vdash lt : (T_0, \ldots, T_n)}{c \ lt : (c : (T_0, \ldots, T_n))}$$

$$\frac{\Gamma \vdash lt : (T_{0,1}, \ldots, T_{0,k_n}) \quad lss : \begin{pmatrix} a_1 : (T_{1,1}, \ldots, T_{1,k_1}), \\ \vdots \\ a_n : (T_{n,1}, \ldots, T_{n,k_n}) \end{pmatrix}}{c \ lt \ \texttt{"|"} \ lss : \begin{pmatrix} a_0 : (T_{0,1}, \ldots, T_{0,k_0}), \\ a_1 : (T_{1,1}, \ldots, T_{1,k_1}), \\ \vdots \\ a_n : (T_{n,1}, \ldots, T_{n,k_n}) \end{pmatrix}}$$

Inference Rules for `<list-statement>`

Judgment: $\Gamma_1, \Delta_1, ls \vdash \Gamma_2, \Delta_2$

$$\frac{\Gamma_1 = \Gamma_2 \quad \Delta_1 = \Delta_2}{\Gamma_1, \Delta_1 \texttt{""} \vdash \Gamma_2, \Delta_2}$$

$$\frac{\Gamma_1, \Delta_1, s \vdash \Gamma_2, \Delta_2 \quad \Gamma_2, \Delta_2, ls \vdash \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1, s \ \texttt{";"} \ ls \vdash \Gamma_3, \Delta_3}$$

Inference Rules for `<maybe-statement-sign>`

Judgment: $\Gamma, mss \vdash a : T$

$$\Gamma, \texttt{""} \vdash a : T$$

$$\frac{\Gamma \vdash t : T a_1 = a_2}{\Gamma, a_1 \ \texttt{":"} \ t \ \texttt{";"} \vdash a_2 : T}$$

Inference Rules for `<statement>`

Judgment: $\Gamma_1, \Delta_1, s \vdash \Gamma_2, \Delta_2$

$$\frac{\Gamma_1 = \Gamma_2 \quad (a, \_) \notin \Delta_1 \qquad \Gamma_1, mss \vdash e : T \quad \Gamma_1, \Delta_1 \vdash e : T \quad \Delta_2 = \Delta_1 \cup \{(a, \overline{\Gamma}(T))\}}{\Gamma_1, \Delta_1, mss\ a\ \texttt{"="} e \vdash \Gamma_2, \Delta_2}$$

$$\frac{\Delta_1 = \Delta_2 \quad (c, \_) \notin \Gamma_1 \quad \Gamma \vdash t : T_1 \qquad T_2 \text{ is a mono type} \quad lsv : (a_1, \ldots, a_n) \quad \{a_1 \ldots a_n\} = \text{free}(T_2) \qquad \forall a_1 \ldots \forall a_n . T_2 = T_1 \quad \Gamma_2 = \Gamma_2 \cup \{(c, T_1)\}}{\Gamma_1, \Delta_1, \texttt{"type alias"}\ c\ lsv\ \texttt{"="}\ t \vdash \Gamma_2, \Delta_2}$$

$$\frac{\begin{array}{c} (c, \_) \notin \Gamma_1 \quad lsv : (a_1, \ldots, a_n) \\ lss : (c_1 : (T_{1,1}, \ldots, T_{1,k_1}), \ldots, c_n : (T_{n,1}, \ldots, T_{n,k_n})) \\ \Delta_1 \cap \{(c_1, \_), \ldots, (c_n, \_)\} = \varnothing \quad \{a_1 \ldots a_n\} = \text{free}(T_2) \\ \mu C . c_1\ T_{1,1}\ \ldots\ T_{1,k_1}\ |\ \ldots\ |\ c_n\ T_{n,1}\ \ldots\ T_{n,k_n} = T_2 \quad \forall a_1 \ldots \forall a_n . T_2 = T_1 \\ \Gamma_1 \cup \{(c, T_1)\} = \Gamma_2 \quad \Delta_1 \cup \left\{\begin{array}{l} (c_1, \overline{\Gamma}(T_{1,1} \to \cdots \to T_{1,k_1} \to T_1)), \\ \vdots \\ (c_n, \overline{\Gamma}(T_{n,1} \to \cdots \to T_{n,k_n} \to T_1)) \end{array}\right\}) = \Delta_2 \end{array}}{\Gamma_1, \Delta_1, \texttt{"type"}\ c\ lsv\ \texttt{"="}\ lss \vdash \Gamma_2, \Delta_2}$$

The list *lss* provides us with the structure of the type. From there we construct the type T_2 and bind all variables, thus creating the poly type T_1. Additionally, every sort c_i for $i \in \mathbb{N}_1^n$ has its own constructor that gets added to Δ_1 under the name c_i. In Elm these constructors are the only constants beginning with an upper-case letter.

Inference Rules for `<maybe-main-sign>`

Judgment: $\Gamma, mms \vdash \text{main} : T$

$$\Gamma, \texttt{""} \vdash \text{main} : T$$

$$\frac{\Gamma \vdash t : T}{\Gamma, \texttt{"main :"}t\texttt{";"} \vdash \text{main} : T}$$

Inference Rules for `<program>`

Judgment: $prog : T$

$$\frac{\varnothing, \varnothing, ls \vdash \Gamma, \Delta \quad \Gamma, mms \vdash \text{main} : T \quad \Gamma, \Delta \vdash e : T}{ls\ mms\ \texttt{"main = "}\ e : T}$$

3.4. Denotational Semantics

We will now expore the semantics of the formal language. To do so, we first define a new kind of context.

Definition 4.1: Variable Context

Let Γ be a type context.

—

The function $\Delta : \mathcal{V} \nrightarrow \bigcup_{T \in \mathcal{T}} \text{value}_\Gamma(T)$ is called a *variable context*.

The semantics of the type signature was already defined in Section 3.3: The semantic of a type signature is its type. We will therefore define the same concept again but now in a denotational style.

Definition 4.2: Type Signature Semantic

Let Let Γ be a type context.

—

We define the following semantic evaluation functions:

$$\begin{aligned}
[\![.]\!]_\Gamma &: \texttt{<list-type-fields>} \to (\mathcal{V} \times \mathcal{T})^* \\
[\![\texttt{""}]\!]_\Gamma &= (\ ) \\
[\![a_0\ \texttt{":"}\ t_0\ \texttt{","}\ ltf]\!]_\Gamma &= ((a_0, T_0), \ldots, (a_n, T_n)) \\
&\qquad \text{such that } T_0 = [\![t_0]\!]_\Gamma \text{ and } [\![ltf]\!]_\Gamma = ((a_1, T_1), \ldots, (a_n, T_n)) \\
&\qquad \text{where } n \in \mathbb{N} \text{ and } T_i \in \mathcal{T}, a_i \in \mathcal{V} \text{ for all } i \in \mathbb{N}_0^n
\end{aligned}$$

$$\begin{aligned}
[\![.]\!]_\Gamma &: \texttt{<list-type>} \to \mathcal{T}^* \\
[\![\texttt{""}]\!]_\Gamma &= (\ ) \\
[\![t_0\ lt]\!]_\Gamma &= (T_0, \ldots, T_n) \text{ and } [\![lt]\!]_\Gamma = (T_1, \ldots, T_n) \\
&\qquad \text{where } n \in \mathbb{N} \text{ and } T_i \in \mathcal{T} \text{ for all } i \in \mathbb{N}_0^n
\end{aligned}$$

$$
\begin{aligned}
[\![.]\!]_\Gamma &: \texttt{<type>} \to \mathcal{T} \\
[\![\texttt{"Bool"}]\!]_\Gamma &= \mathit{Bool} \\
[\![\texttt{"Int"}]\!]_\Gamma &= \mathit{Int} \\
[\![\texttt{"List"}\ t]\!]_\Gamma &= \mathit{List}\ T \\
&\quad \text{such that } T = [\![t]\!]_\Gamma \\
&\quad \text{where } T \in \mathcal{T} \\
[\![\texttt{"("}\ t_1\ \texttt{","}\ t_2\ \texttt{")"}]\!]_\Gamma &= (T_1, T_2) \\
&\quad \text{such that } T_1 = [\![t_1]\!]_\Gamma \text{ and } T_2 = [\![t_2]\!]_\Gamma \\
&\quad \text{where } T_1, T_2 \in \mathcal{T} \\
[\![\texttt{"\{"}\ \mathit{ltf}\ \texttt{"\}"}]\!]_\Gamma &= \{a_1 : T_1, \ldots, a_n : T_n\} \\
&\quad \text{such that } [\![\mathit{ltf}]\!]_\Gamma = ((a_1, T_1), \ldots, (a_n, T_n)) \\
&\quad \text{where } n \in \mathbb{N} \text{ and } T_i \in \mathcal{T}, a_i \in \mathcal{V} \text{ for all } i \in \mathbb{N}_0^n \\
[\![t_1\ \texttt{"->"}\ t_2]\!]_\Gamma &= T_1 \to T_2 \\
&\quad \text{such that } [\![t_1]\!]_\Gamma = T_1 \text{ and } [\![t_2]\!]_\Gamma = T_2 \\
[\![c\ \mathit{lt}]\!]_\Gamma &= \overline{T}\ T_1 \ldots T_n \\
&\quad \text{such that } (c, T) \in \Gamma \text{ and } (T_1, \ldots, T_n) = [\![\mathit{lt}]\!]_\Gamma \\
&\quad \text{where } n \in \mathbb{N}, T \in \mathcal{T} \text{ and } T_i \in \mathcal{T} \text{ for all } i \in \mathbb{N}_1^n \\
[\![a]\!]_\Gamma &= \forall b.b
\end{aligned}
$$

An Elm program is nothing more than an expression with some syntax sugar around it. Semantics of an expression is therefore the heart piece of this section.

Definition 4.3: Expression Semantic

Let Γ be a type context and let Δ be variable contexts.

—

We define the following semantic evaluation functions:

$$[\![.]\!]_{\Gamma,\Delta} : \texttt{<list-exp-field>} \to (\mathcal{V} \times \bigcup_{T \in \mathcal{T}} \text{value}_\Gamma(T))^*$$

$$\begin{aligned}
[\![a \texttt{ "=" } e]\!]_{\Gamma,\Delta} &= \{a = s_2\} \\
&\quad \text{such that } s_2 = [\![e]\!]_{\Gamma,\Delta} \\
&\quad \text{where } s_2 \in \text{value}_\Gamma(T) \text{ for } T \in \mathcal{T} \\
[\![a_1 \texttt{ "=" } e \texttt{ "," } lef]\!]_{\Gamma,\Delta} &= \{a_1 = s_1, \ldots, a_n = s_n\} \\
&\quad \text{such that } \{a_1 = s_1\} = [\![a \texttt{ "=" } e]\!]_{\Gamma,\Delta} \\
&\quad \text{and } \{a_2 = s_2, \ldots, a_n = s_n\} = [\![lef]\!]_{\Gamma,\Delta} \\
&\quad \text{where } n \in \mathbb{N} \text{ and } a_i \in \mathcal{V}, s_i \in \text{value}_\Gamma(T_i) \\
&\quad \text{for } T_i \in \mathcal{T} \text{ for } i \in \mathbb{N}_0^n
\end{aligned}$$

$$[\![.]\!] : \texttt{<maybe-exp-sign>} \to (\ )$$

$$\begin{aligned}
[\![\texttt{""}]\!] &= (\ ) \\
[\![a \texttt{ ":" } t \texttt{ ";"}]\!] &= (\ )
\end{aligned}$$

$$[\![.]\!] : \texttt{<bool>} \to \text{value}_\varnothing(\mathit{Bool})$$

$$[\![b]\!] = \begin{cases} \mathit{True} & \text{if } b = \texttt{"True"} \\ \mathit{False} & \text{if } b = \texttt{"False"} \end{cases}$$

$$[\![.]\!] : \texttt{<int>} \to \text{value}_\varnothing(\mathit{Int})$$

$$\begin{aligned}
[\![\texttt{"0"}]\!] &= 0 \\
[\![\texttt{"-"} \ nr]\!] &= \mathit{Neg}\ \mathit{Succ}^{nr}\ 0 \\
[\![nr]\!] &= \mathit{Pos}\ \mathit{Succ}^{nr}\ 0
\end{aligned}$$

$$[\![.]\!]_{\Gamma,\Delta} : \texttt{<list-exp>} \to \bigcup_{T \in \mathcal{T}} \text{value}_\Gamma(T))^*$$

$$\begin{aligned}
[\![\texttt{""}]\!]_{\Gamma,\Delta} &= \mathit{Empty} \\
[\![e \texttt{ "," } le]\!]_{\Gamma,\Delta} &= \mathit{Cons}\ s_1\ s_2 \\
&\quad \text{such that } s_1 = [\![e]\!]_{\Gamma,\Delta} \wedge s_2 = [\![le]\!]_{\Gamma,\Delta} \\
&\quad \text{where } n \in \mathbb{N} \text{ and } s_i \in \text{value}_\Gamma(T_i), T_i \in \mathcal{T} \text{ for each } i \in \mathbb{N}_0^n
\end{aligned}$$

$$[\![.]\!]_{\Gamma,\Delta} : \texttt{<exp>} \to \bigcup_{T \in \mathcal{T}} \text{value}_\Gamma(T)$$

$$[\![\texttt{"foldl"}]\!]_{\Gamma,\Delta} = s$$

$$\text{where } s = \lambda f.\lambda e_1.\lambda l_1. \begin{cases} e_1 & \text{if } [\,] = l_1 \\ f(e_2, s(f, e_1, l_2)) & \text{if } \mathit{Cons}\ e_2\ l_2 = l_1 \end{cases}$$

$$\text{and } e_1 \in \text{value}_\Gamma(T_1), e_2 \in \text{value}_\Gamma(T_2)$$

$$\text{and } l_1, l_2 \in \text{value}_\Gamma(\mathit{List}\ T_2)$$

$$\text{and } f \in \text{value}_\Gamma(T_2 \to T_1 \to T_1) \text{ for } T_1, T_2 \in \mathcal{T}$$

$$\text{and } s \in \bigcup_{T \in \mathcal{T}} \text{value}_\Gamma(T)$$

$$[\![\texttt{"(::)"}]\!]_{\Gamma,\Delta} = \lambda e.\lambda l.\mathit{Cons}\ e\ l$$

$$\text{where } e \in \text{value}_\Gamma(T)$$

$$\text{and } l \in \text{value}_\Gamma(\mathit{List}\ T)$$

$$\text{for } T \in \mathcal{T}$$

$$[\![\texttt{"(+)"}]\!]_{\Gamma,\Delta} = \lambda n.\lambda m.n + m$$

$$\text{where } n, m \in \mathbb{Z}$$

$$[\![\texttt{"(-)"}]\!]_{\Gamma,\Delta} = \lambda n.\lambda m.n - m$$

$$\text{where } n, m \in \mathbb{Z}$$

$$[\![\texttt{"(*)"}]\!]_{\Gamma,\Delta} = \lambda n.\lambda m.n * m$$

$$\text{where } n, m \in \mathbb{Z}$$

$$[\![\texttt{"(//)"}]\!]_{\Gamma,\Delta} = \lambda n.\lambda m. \begin{cases} \left\lfloor \frac{n}{m} \right\rfloor & \text{if } m \neq 0 \\ 0 & \text{else} \end{cases}$$

$$\text{where } n, m \in \mathbb{Z}$$

$$[\![\texttt{"(<)"}]\!]_{\Gamma,\Delta} = \lambda n.\lambda m.n < m$$

$$\text{where } n, m \in \mathbb{Z}$$

$$[\![\texttt{"(==)"}]\!]_{\Gamma,\Delta} = \lambda n.\lambda m.(n = m)$$

$$\text{where } n, m \in \mathbb{Z}$$

$$[\![\texttt{"not"}]\!]_{\Gamma,\Delta} = \lambda b.\neg b$$

$$\text{where } b \in \text{value}_\Gamma(\mathit{Bool})$$

$$[\![\texttt{"(\&\&)"}]\!]_{\Gamma,\Delta} = \lambda b_1.\lambda b_2.b_1 \wedge b_2$$

$$\text{where } b_1, n_2 \in \text{value}_\Gamma(\mathit{Bool})$$

$$[\![\texttt{"(||)"}]\!]_{\Gamma,\Delta} = \lambda b_1.\lambda b_2.b_1 \vee b_2$$
$$\text{where } b_1, n_2 \in \text{value}_\Gamma(\mathit{Bool})$$

$$\left[\!\!\left[\begin{matrix}\texttt{"if"}\ e_1\ \texttt{"then"} \\ e_2\ \texttt{"else"}\ e_3\end{matrix}\right]\!\!\right]_{\Gamma,\Delta} = \begin{cases}[\![e_2]\!]_{\Gamma,\Delta} & \text{if } b \\ [\![e_3]\!]_{\Gamma,\Delta} & \text{if } \neg b\end{cases}$$
$$\text{such that } b = [\![e_1]\!]_{\Gamma,\Delta}$$
$$\text{where } b \in \text{value}(\mathit{Bool})$$

$$[\![\texttt{"\{"}\ \mathit{lef}\ \texttt{"\}"}]\!]_{\Gamma,\Delta} = [\![\mathit{lef}]\!]_{\Gamma,\Delta}$$

$$[\![\texttt{"\{ \}"}]\!]_{\Gamma,\Delta} = \{\}$$

$$[\![\texttt{"\{"}\ a\ \texttt{"|"}\ \mathit{lef}\ \texttt{"\}"}]\!]_{\Gamma,\Delta} = \{a_1 = s_1, \dots, a_m = s_m\}$$
$$\text{such that } \{a_1 = s_1, \dots, a_n = s_n\} = [\![\mathit{lef}]\!]_{\Gamma,\Delta}$$
$$\text{and } (a, \left\{\begin{matrix} a_1 = \_, \dots, a_n = \_, \\ a_{n+1} = s_{n+1}, \dots, a_m = s_m \end{matrix}\right\}) \in \Delta$$
$$\text{where } n, m \in \mathbb{N} \text{ such that } n \leq m \text{ and } a_i \in \mathcal{V},$$
$$s_i \in \text{value}(T_i), T_i \in \mathcal{T} \text{ for } i \in \mathbb{N}_0^m$$

$$[\![a_0\ \texttt{"."}\ a_1]\!]_{\Gamma,\Delta} = s'$$
$$\text{such that } \Delta(a_0) = \{a_1 : s', \dots\}$$
$$\text{where } s' \in \text{value}(T) \text{ for } T \in \mathcal{T}$$

$$\left[\!\!\left[\begin{matrix}\texttt{"let"}\ \mathit{mes}\ a\ \texttt{"="}\ e_1 \\ \texttt{"in"}\ e_2\end{matrix}\right]\!\!\right]_{\Gamma,\Delta} = [\![e_2]\!]_{\Gamma,\Delta\cup\{(a,s')\}}$$
$$\text{such that } s' = [\![e_1]\!]_{\Gamma,\Delta}$$
$$\text{where } s' \in \text{value}(T) for T \in \mathcal{T}$$

$$[\![e_1\ e_2]\!]_{\Gamma,\Delta} = s_1(s_2)$$
$$\text{such that } s_1 = [\![e_1]\!]_{\Gamma,\Delta} \text{ and } s_2 = [\![e_2]\!]_{\Gamma,\Delta}$$
$$\text{where } s_1 \in \text{value}_\Gamma(T_1 \to T_2) \text{ and } s_2 \in \text{value}_\Gamma(T_1) \text{for } T_1, T_2 \in \mathcal{T}$$

$$[\![b]\!]_{\Gamma,\Delta} = [\![b]\!]$$

$$[\![i]\!]_{\Gamma,\Delta} = [\![i]\!]$$

$$[\![\texttt{"["}\ \mathit{le}\ \texttt{"]"}]\!]_{\Gamma,\Delta} = [s_1, \dots, s_n]$$
$$\text{such that } (s_1, \dots, s_n) = [\![\mathit{le}]\!]_{\Gamma,\Delta}$$
$$\text{where } n \in \mathbb{N} \text{ and } s_i \in \text{value}_\Gamma(T) \text{ for } T \in \mathcal{T}$$

$$
\begin{aligned}
[\![\texttt{"("}\ e_1\ \texttt{","}\ e_2\ \texttt{")"}]\!]_{\Gamma,\Delta} &= (s_1, s_2) \\
&\quad \text{such that } s_1 = [\![e_1]\!] \text{ and } s_2 = [\![e_2]\!] \\
&\quad \text{where } s_1 \in \text{value}_\Gamma(T_1) \text{ and } s_2 \in \text{value}_\Gamma(T_1) \\
[\![\texttt{"\textbackslash"}\ a\ \texttt{"->"}\ e]\!]_{\Gamma,\Delta} &= \lambda b.[\![e]\!]_{\Gamma,\Delta\cup\{(a,b)\}} \\
&\quad \text{where } b \in \mathcal{V} \\
[\![c]\!]_{\Gamma,\Delta} &= s \text{ such that } (c, s) \in \Delta \\
&\quad \text{where } s \in \bigcup_{T\in\mathcal{T}} \text{value}_\Gamma(T) \\
[\![a]\!]_{\Gamma,\Delta} &= s \text{ such that } (a, s) \in \Delta \\
&\quad \text{where } s \in \bigcup_{T\in\mathcal{T}} \text{value}_\Gamma(T)
\end{aligned}
$$

Statements are, semantically speaking, just functions that either map the type- or variable-context.

Definition 4.4: Statement Semantic

We define the following semantic evaluation functions:

$$
\begin{aligned}
[\![.]\!] &: \texttt{<list-statement-var>} \to \mathcal{V}^* \\
[\![\texttt{""}]\!] &= (\ ) \\
[\![a_0\ \ lsv]\!] &= (a_0, \ldots, a_n) \\
&\quad \text{such that } (a_1, \ldots, a_n) = [\![lsv]\!] \\
&\quad \text{where } \ n \in \mathbb{N} \text{ and } a_i \in \mathcal{V} \text{ for } i \in \mathbb{N}_0^n
\end{aligned}
$$

$$
\begin{aligned}
[\![.]\!] &: \texttt{<list-statement>} \to ((\mathcal{V} \nrightarrow \mathcal{T}) \times (\mathcal{V} \nrightarrow \mathcal{S})) \to ((\mathcal{V} \nrightarrow \mathcal{T}) \times (\mathcal{V} \nrightarrow \mathcal{S})) \\
[\![\texttt{""}]\!] &= id \\
[\![st\ \texttt{","}\ ls]\!] &= g \circ f \\
&\quad \text{such that } f = [\![st]\!] \text{ and } g = [\![ls]\!] \\
&\quad \text{where } f, g \in ((\mathcal{V} \nrightarrow \mathcal{T}) \times (\mathcal{V} \nrightarrow \mathcal{S})) \to ((\mathcal{V} \nrightarrow \mathcal{T}) \times (\mathcal{V} \nrightarrow \mathcal{S}))
\end{aligned}
$$

$$
\begin{aligned}
[\![.]\!] &: \texttt{<maybe-statement-sign>} \to (\ ) \\
[\![\texttt{""}]\!] &= (\ ) \\
[\![a\ \texttt{":"}\ t\ \texttt{";"}]\!] &= (\ )
\end{aligned}
$$

$$[\![.]\!] : \texttt{<statement>} \to ((\mathcal{V} \nrightarrow \mathcal{T}) \times (\mathcal{V} \nrightarrow \mathcal{S})) \to ((\mathcal{V} \nrightarrow \mathcal{T}) \times (\mathcal{V} \nrightarrow \mathcal{S}))$$

$$[\![mss\ a\ \texttt{"="}\ e]\!](\Gamma, \Delta) = (\Gamma, \Delta \cup \{(a, s')\})$$

such that $s' = [\![e]\!]_{\Gamma,\Delta}$

where $s' \in \text{value}(T)$ for $T \in \mathcal{T}$

$$\left[\!\!\left[\begin{matrix}\texttt{"type alias"} \\ c\ lsv\ \texttt{"="}\ t\end{matrix}\right]\!\!\right](\Gamma, \Delta) = (\Gamma \cup \{(c, T)\}, \Delta)$$

such that $T = [\![\texttt{t}]\!]_\Gamma$

$$[\![.]\!] : \texttt{<maybe-main-sign>} \to (\ )$$

$$[\![\texttt{""}]\!] = (\ )$$

$$[\![\texttt{"main : "}\ t\ \texttt{";"}]\!] = (\ )$$

$$[\![.]\!] : \texttt{<program>} \to \bigcup_{T \in \mathcal{T}} \text{value}_\varnothing(T)$$

$$[\![ls\ mms\ \texttt{"main = "}\ e]\!] = [\![e]\!]_{\Gamma,\Delta}$$

such that $(\Gamma, \Delta) = [\![ls]\!](\varnothing, \varnothing)$

where Γ is a type context and Δ is a variable context.

3.5. Soundness of the Inference Rules

In this section we prove the soundness of the inference rules with respect to the denotational semantics. This means we ensure that if we can infer the well-typedness of a program, the execution of the program yields those kinds of values predicted by the inference rules.

3.5.1. Soundness of the Type Signature

The inference rules and the semantics for the type signatures are built in a structurally similar way. Thus, we will now show that the semantics of a phrase yields the kind of result predicted by the inference rules.

Theorem 5.1

Let Γ be a type context, $ltf \in$ `<list-type-fields>`, $a_i \in \mathcal{V}$,$T_i \in \mathcal{T}$ for $i \in \mathbb{N}_1^n$ and $n \in \mathbb{N}_0$. Assume that $\Gamma \vdash ltf : \{a_1 : T_1, \ldots, a_n : T_n\}$ can be derived.

—

Then $[\![ltf]\!]_\Gamma = ((a_1, T_1), \ldots, (a_n, T_n))$.

Proof. Let Γ be a type context, $ltf \in$ `<list-type-fields>`, $a_i \in \mathcal{V}$,$T_i \in \mathcal{T}$ for $i \in \mathbb{N}_1^n$ and $n \in \mathbb{N}_0$. Assume $ltf : \{a_1 : T_1, \ldots, a_n : T_n\}$ can be derived.

- **Case** $ltf =$ `""` for $n = 0$: Then $[\![\texttt{ltf}]\!] = \{\}$ and therefore the conclusion holds.
- **Case** $ltf = a_1$ `":"` T_1 `","` ltf_1 for $ltf_1 \in$ `<list-type-field>`: Then by the premise of the inference rule for ltf we can assume that $\Gamma \vdash ltf_1 : \{a_2 : T_2, \ldots, a_n : T_n\}$ can be derived and by induction hypothesis $[\![ltf_1]\!]_\Gamma = ((a_2, T_2), \ldots, (a_n, T_n))$. We can now use the semantics as describe in its definition: $[\![ltf]\!]_\Gamma = [\![a_1 \texttt{ ":" } T_1 \texttt{ "," } ltf_1]\!] = ((a_1, T_1), \ldots, (a_n, T_n))$. Thus the conclusion $[\![ltf]\!]_\Gamma = ((a_1, T_1), \ldots, (a_n, T_n))$ follows.

□

Theorem 5.2

Let Γ be a type context, $lt \in$ `<list-type>`$, T_i \in \mathcal{T}$ for $i \in \mathbb{N}_1^n$ and $n \in \mathbb{N}_0$. Assume $\Gamma \vdash lt : (T_1, \ldots, T_n)$ can be derived.

—

Then $[\![lt]\!]_\Gamma = (T_1, \ldots, T_n)$.

Proof. See the combined proof of the conjunction of Theorem 5.2 and 5.3 below. □

Theorem 5.3

Let Γ be a type context, $t \in$ `<type>` and $T \in \mathcal{T}$. Assume $\Gamma \vdash t : T$ can be derived.

—

Then $[\![t]\!]_\Gamma = T$.

Proof. Combined proof of Theorems 5.2 and 5.3.

We prove the conjunction of Theorem 5.2 and 5.3 by simultaneous induction over the structure of the mutually recursive grammar rules for `<list-type>` and `<type>`.

Let Γ be a type context, $lt \in$ `<list-type>`$, T_i \in \mathcal{T}$ for $i \in \mathbb{N}_1^n$ and $n \in \mathbb{N}_0$. Assume $\Gamma \vdash lt : (T_1, \ldots, T_n)$ can be derived. We show $[\![lt]\!]_\Gamma = (T_1, \ldots, T_n)$.

- **Case** $lt =$ `""` for $n = 0$: Then $[\![lt]\!]_\Gamma = (\ )$ and thus the conclusion holds.
- **Case** $lt = t_1\ lt_1$ for $t_1 \in$ `<type>` for $lt_1 \in$ `<list-type>`: Then from the premise of the inference rule, we assume that $\Gamma \vdash lt_1 : (T_2, \ldots, T_n)$ and $\Gamma \vdash t_1 : T_1$ hold. The assumption of Theorem 5.3, namely that $\Gamma \vdash t_1 : T_1$ can be derived, now

holds. By its induction hypothesis we can therefore conclude that $[\![t_1]\!]_\Gamma = T_1$ for $T_1 \in \mathcal{T}$. The assumption of Theorem 5.2, namely $\Gamma \vdash lt_1 : (T_2, \ldots, T_n)$, holds and therefore by the induction hypothesis of Theorem 5.2 we obtain $[\![t_1 \;\; lt_1]\!]_\Gamma = (T_1, T_2, \ldots, T_n)$. Thus the conclusion $[\![lt]\!]_\Gamma = (T_1, \ldots, T_n)$ holds.

Let Γ be a type context, $t \in$ `<type>` and $T \in \mathcal{T}$. Assume $\Gamma \vdash t : T$ can be derived. We show $[\![t]\!]_\Gamma = T$.

- **Case** $t =$ `"Bool"`: Then $[\![t]\!]_\Gamma = \mathit{Bool}$ and the conclusion holds.
- **Case** $t =$ `"Int"`: Then by the premise of the inference rule for `"Int"`, we can assume that $\Gamma \vdash t : \mathit{Int}$ can be derived and therefore $[\![t]\!]_\Gamma = \mathit{Int}$. We see that the conclusion holds.
- **Case** $t =$ `"List"` t_1, for $t_1 \in$ `<type>`: By the premise of the inference rule we assume $\Gamma \vdash t_1 : T_1$ can be derived and by induction hypothesis $[\![t_1]\!]_\Gamma = T_1$ for given $T_1 \in \mathcal{T}$. Then $[\![t]\!]_\Gamma = \mathit{List}\ T_1$. Thus the conclusion holds.
- **Case** $t =$ `"("` t_1 `","` t_2 `")"`, for $t_1, t_2 \in$ `<type>`: By the premise of the inference rule $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$ hold for given $T_1, T_2 \in \mathcal{T}$. Then by induction hypothesis $[\![t_1]\!]_\Gamma = T_1$ and $[\![t_2]\!]_\Gamma = T_2$. Thus by the definition of the semantics the conclusion holds analogously to the cases above.
- **Case** $t =$ `"{"` ltf `"}"`, for $\mathit{ltf} \in$ `<list-type-field>`: Then by the premise of the inference rule $\Gamma \vdash \mathit{ltf} : \{a_1 : T_1, \ldots, a_n : T_n\}$ for $a_i \in \mathcal{V}$, $T_i \in \mathcal{T}$, $i \in \mathbb{N}_1^n$ and $n \in \mathbb{N}_0$. Thus by Theorem 5.1 $[\![\mathit{ltf}]\!]_\Gamma = ((a_1, T_1), \ldots, (a_n, T_n))$ and therefore the conclusion holds analogously to the cases above.
- **Case** $t = t_1$ `"->"` t_2, for $t_1, t_2 \in$ `<type>`: By the premise of the inference rule $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$ hold for given $T_1, T_2 \in \mathcal{T}$. By induction hypothesis $[\![t_i]\!]_\Gamma = T_i$ for $i \in \{1, 2\}$. Thus by the definition of the semantics the conclusion holds analogously to the cases above.
- **Case** $t = c\ lt$ for $lt \in$ `<list-type>` and $c \in$ `<upper-var>`: By the premise of the inference rule we know $(c, T') \in \Gamma$ with $T' \in \mathcal{T}$ and can assume that $\Gamma \vdash lt : (T_1, \ldots, T_n)$ for $T_i \in \mathcal{T}$, $i \in \mathbb{N}^n$ and $n \in \mathbb{N}_0$ can be derived. Therefore, the assumption of Theorem 5.2, namely that $\Gamma \vdash lt : (T_1, \ldots, T_n)$ can be derived, holds and by applying its induction hypothesis, we know $[\![lt]\!]_\Gamma = (T_1, \ldots, T_n)$. Thus by the definition of the semantics the conclusion holds.
- **Case** $t = a$ for $a \in \mathcal{V}$: Then by the definition of the semantics the conclusion holds analogously to the cases above.

□

3.5.2. Soundness of the Variable Context

In Section 3.3 we said that Δ is a type context, where as in Section 4.2.6 we said that Δ is a variable context. We will now define the relation between the two.

Definition 5.1: Similar Variable context

Let Γ, Δ be type contexts and Δ' a variable context.

—

We say Δ' is *similar to* Δ *with respect to* Γ if and only if for all $T \in \mathcal{T}$ and for all $a \in \mathcal{V}$ the following holds:

$$(a, T) \in \Delta \Rightarrow \exists e \in \text{value}_\Gamma(\overline{\Gamma}(T)).(a, e) \in \Delta'.$$

Theorem 5.4

Let Γ, Δ be type contexts and Δ' be a variable context similar to Δ with respect to Γ. Let $a \in \mathcal{V}$ and $T \in \mathcal{T}$. Let $e \in \text{value}_\Gamma(\overline{\Gamma}(T))$.

—

Then $\Delta' \cup \{(a, e)\}$ is similar to $\Delta \cup \{(a, \overline{\Gamma}(T))\}$ with respect to Γ.

Proof. Let Δ be a type context and Δ' be a variable context similar to Δ with respect to Γ. Let $a \in \mathcal{V}$ and $T \in \mathcal{T}$. Let $e \in \text{value}_\Gamma(\overline{\Gamma}(T))$.

We know Δ is similar to Δ' with respect to Γ, meaning for all $T' \in \mathcal{T}$ and for all $a' \in \mathcal{V}$ the following holds:

$$(a', T') \in \Delta \Rightarrow \exists d \in \text{value}_\Gamma(\overline{\Gamma}(T)) \text{ such that } (a', d) \in \Delta'.$$

Let $a' \in \mathcal{V}$ and $T' \in \mathcal{T}$ such that $(a', T') \in \Delta \cup \{(a, \overline{\Gamma}(T))\}$.

- **Case** $(a', T') \in \Delta$: Because Δ is similar to Δ' we can directly conclude $\exists d \in \text{value}_\Gamma(\overline{\Gamma}(T))$ such that $(a', d) \in \Delta' \cup \{(a, e)\}$.
- **Case** $(a', T') = (a, \overline{\Gamma}(T))$: We know $e \in \text{value}_\Gamma(\overline{\Gamma}(T))$ and $(a, e) \in \Delta' \cup \{(a, e)\}$. By $a' = a$ we therefore conclude $(a', e) \in \Delta' \cup \{(a, e)\}$.

□

Types in Δ are all the most generalized types. Instead of proving this, we show that the semantics only produces values of most generalized types. This is a weaker statement but strong enough for our purposes.

3.5.3. Soundness of the Expression Rules

We can now use the definition of well-formed variable contexts, to prove the soundness of the rules for inferring the types of expressions.

Theorem 5.5

Let $b \in$ `<bool>`.

—

Then $[\![b]\!] \in \text{value}_{\varnothing}(\mathit{Bool})$.

Proof. Let $b \in$ `<bool>`.

- **Case** $b =$ `"True"`: Then $[\![\texttt{b}]\!] = \mathit{True}$. Thus the conclusion holds.
- **Case** $b =$ `"False"`: Then $[\![\texttt{b}]\!] = \mathit{False}$. Thus the conclusion holds.

□

Theorem 5.6

Let $i \in$ `<int>`.

—

Then $[\![i]\!] \in \text{value}_{\varnothing}(\mathit{Int})$.

Proof. Let $i \in$ `<int>`.

- **Case** $i =$ `"0"`: Then $[\![\texttt{i}]\!] = 0$. Thus the conclusion holds.
- **Case** $i = n$ for $n \in \mathbb{N}$: Then $[\![\texttt{i}]\!] = \mathit{Succ}^n\ 0$. Thus the conclusion holds.
- **Case** $i =$ `"-"` n for $n \in \mathbb{N}$: Then $[\![\texttt{i}]\!] = \mathit{Neg}\ \mathit{Succ}^n\ 0$. Thus the conclusion holds.

□

Theorem 5.7

Let Γ,Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ and $\mathit{lef} \in$ `<list-exp-field>`. Assume $\Gamma, \Delta \vdash \mathit{lef} : T$ can be derived for $T = \{a_1 : T_1, \ldots, a_n : T_n\} \in \mathcal{T}$, $a_i \in \mathcal{V}, T_i \in \mathcal{T}$, for all $i \in \mathbb{N}_1^n$, and $n \in \mathbb{N}_0$.

—

Then $[\![\mathit{lef}]\!]_{\Gamma,\Delta'} \in \text{value}_{\Gamma}(\overline{\Gamma}(T))$.

Proof. See the combined proof of the conjunction of Theorem 5.7, 5.8 and 5.9 below. □

Theorem 5.8

Let Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ and $le \in$ `<list-exp>`. Assume $\Gamma, \Delta \vdash le : List\ T$ can be derived for $T \in \mathcal{T}$.

—

Then $[\![le]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(List\ T))$.

Proof. See the combined proof of the conjunction of Theorem 5.7, 5.8 and 5.9 below. □

Theorem 5.9

Let Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ. Let $e \in$ `<exp>` and $T \in \mathcal{T}$. Assume $\Delta, \Gamma \vdash e : T$ can be derived.

—

Then $[\![e]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T))$.

Proof. We prove the conjunction of Theorem 5.7, 5.8 and 5.9 by simultaneous induction over the structure of the mutually recursive grammar rules for `<list-exp-field>`, `<list-exp>` and `<exp>`.

Let Γ,Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ and $lef \in$ `<list-exp-field>`. Assume the judgment $\Gamma, \Delta \vdash lef : T$ can be derived for $T = \{a_1 : T_1, \ldots, a_n : T_n\} \in \mathcal{T}$, $a_i \in \mathcal{V}, T_i \in \mathcal{T}$, for all $i \in \mathbb{N}_1^n$ and given $n \in \mathbb{N}_0$. We show $[\![lef]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T))$.

- **Case** $lef = a_1$ `"="` e_1 for $e_1 \in$ `<exp>` and $n = 1$: Then by the premise of the inference rule we assume $\Gamma, \Delta \vdash e_1 : T_1$ can be derived and therefore the assumption of Theorem 5.9 holds. By applying said theorem we can therefore conclude $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1))$. Then $[\![lef]\!]_{\Gamma,\Delta'} = \{a_1 = s_1\}$ for $s_1 = [\![e_1]\!]_{\Gamma,\Delta'}$ and therefore the conclusion holds.

- **Case** $lef = a_1$ `"="` e_1 `","` lef_0 for $e_1 \in$ `<exp>` and $lef_0 \in$ `<list-exp-field>`: Then by the premise of the inference rule we assume $\Gamma, \Delta \vdash lef_0 : \{a_2 : T_2, \ldots, a_n : T_n\}$. Then $\Gamma, \Delta \vdash e_1 : T_1$ can both be derived. Thus the assumption of Theorem 5.9 holds and by the induction hypothesis of said theorem $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1))$. By $\Gamma, \Delta \vdash lef_0 : T$ the assumption for the induction hypothesis

of Theorem 5.7 holds and therefore by appling the theorem we obtain $[\![lef_0]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(\{a_2 : T_2, \ldots, a_n : T_n\}))$. Then $[\![lef]\!]_{\Gamma,\Delta'} = \{a_1 = s_1, \ldots, a_n = s_n\}$ for $s_i = [\![e_i]\!]_{\Gamma,\Delta'}$ for $i \in \mathbb{N}_1^n$ and thus the conclusion holds.

Let Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ and $le \in$ `<list-exp>`. Assume $\Gamma, \Delta \vdash le : \mathit{List}\ T$ can be derived for given $T \in \mathcal{T}$. We show $[\![le]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\mathit{List}\ T)$.

- **Case** $le =$ `""`: Then $[\![\texttt{""}]\!]_{\Gamma,\Delta'} = \mathit{Empty}$ and thus the conclusion holds.
- **Case** $le = e$ `","` le_1 for $e \in$ `<exp>` and $le_1 \in$ `<list-exp>`: Then by the premise of the inference rule we assume $\Gamma, \Delta \vdash e : T$ and $\Gamma, \Delta \vdash le_1 : \mathit{List}\ T$ can be derived. The assumption of Theorem 5.9, namely that $\Gamma, \Delta \vdash e : T$ can be derived, holds and by appling that theorem $[\![e]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T))$. The assumption of Theorem 5.8, namely that $\Gamma, \Delta \vdash le_1 : T$ can be derived, also holds and by appling said theorem we conclude $[\![le_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(\mathit{List}\ T))$. By using the definition of the semantics $[\![le]\!]_{\Gamma,\Delta'} = \mathit{Cons}\ e\ [\![le_1]\!]_{\Gamma,\Delta'}$ and therefore the conclusion holds.

Let Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ. Let $e \in$ `<exp>` and $T \in \mathcal{T}$. Assume $\Delta, \Gamma \vdash e : T$ can be derived.

- **Case** $e =$ `"foldl"`: Then $T = \forall a.\forall b.(a \to b \to b) \to b \to \mathit{List}\ a \to b$ and $[\![\texttt{"foldl"}]\!]_{\Gamma,\Delta'} = s$ for

$$s = \lambda f.\lambda e_1.\lambda l_1. \begin{cases} e_1 & \text{if } [\,] = l_1 \\ f(e_2, s(f, e_1, l_2)) & \text{if } \mathit{Cons}\ e_2\ l_2 = l_1 \end{cases}$$

  where $e_1 \in \text{value}_\Gamma(T_1), e_2 \in \text{value}_\Gamma(T_2)$ and $l_1, l_2 \in \text{value}_\Gamma(\mathit{List}\ T_2)$ and $f \in \text{value}_\Gamma(T_2 \to T_1 \to T_1)$ for $T_1, T_2 \in \mathcal{T}$ and thus the conclusion holds.

- **Case** $e =$ `"(::)"`: Then $T = \forall a.a \to \mathit{List}\ a \to \mathit{List}\ a$ and $[\![\texttt{"(::)"}]\!]_{\Gamma,\Delta'} = \lambda e.\lambda l.\mathit{Cons}\ e\ l$ where $e \in \text{value}_\Gamma(T')$ and $l \in \text{value}_\Gamma(\mathit{List}\ T')$ for $T' \in \mathcal{T}$ and thus the conclusion holds.
- **Case** $e =$ `"(+)"`: Then $T = \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ and $[\![\texttt{"(+)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.n + m$ where $n, m \in \mathbb{Z}$ and thus the conclusion holds.
- **Case** $e =$ `"(-)"`: Then $T = \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ and $[\![\texttt{"(-)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.n - m$ where $n, m \in \mathbb{Z}$ and thus the conclusion holds.
- **Case** $e =$ `"(*)"`: Then $T = \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ and $[\![\texttt{"(*)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.n * m$ where $n, m \in \mathbb{Z}$ and thus the conclusion holds.

- **Case** $e = \texttt{"(//)"}$: Then $T = \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ and

$$[\![\texttt{"(//)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m. \begin{cases} \left\lfloor \frac{n}{m} \right\rfloor & \text{if } m \neq 0 \\ 0 & \text{else} \end{cases}$$

  where $n, m \in \mathbb{Z}$ and thus the conclusion holds.

- **Case** $e = \texttt{"(<)"}$: Then $T = \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$ and $[\![\texttt{"(<)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.n < m$ where $n, m \in \mathbb{Z}$ and thus the conclusion holds.

- **Case** $e = \texttt{"(==)"}$: Then $T = \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$ and $[\![\texttt{"(==)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.(n = m)$ where $n, m \in \mathbb{Z}$ and thus the conclusion holds.

- **Case** $e = \texttt{"not"}$: Then $T = \mathit{Bool} \to \mathit{Bool}$ and $[\![\texttt{"not"}]\!]_{\Gamma,\Delta'} = \lambda b.\neg b$ where $b \in \text{value}_\Gamma(\mathit{Bool})$ and thus the conclusion holds.

- **Case** $e = \texttt{"(\&\&)"}$: Then $T = \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$ and $[\![\texttt{"(\&\&)"}]\!]_{\Gamma,\Delta'} = \lambda b_1.\lambda b_2.b_1 \wedge b_2$ where $b_1, n_2 \in \text{value}_\Gamma(\mathit{Bool})$ and thus the conclusion holds.

- **Case** $e = \texttt{"(||)"}$: Then $T = \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$ and $[\![\texttt{"(||)"}]\!]_{\Gamma,\Delta'} = \lambda b_1.\lambda b_2.b_1 \vee b_2$ where $b_1, n_2 \in \text{value}_\Gamma(\mathit{Bool})$ and thus the conclusion holds.

- **Case** $e = \texttt{"if"}\ e_1\ \texttt{"then"}\ e_2\ \texttt{"else"}\ e_3$ for $e_1, e_2, e_3 \in \texttt{<exp>}$: By the premise of the inference rule we assume $\Gamma, \Delta \vdash e_1 : \mathit{Bool}$, $\Gamma, \Delta \vdash e_2 : T$ and $\Gamma, \Delta \vdash e_3 : T$ can be derived. By induction hypothesis of Theorem 5.9 we know $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}(\mathit{Bool})$, $[\![e_2]\!]_{\Gamma,\Delta'} \in \text{value}(\overline{\Gamma}(T))$ and $[\![e_3]\!]_{\Gamma,\Delta'} \in \text{value}(\overline{\Gamma}(T))$. Thus, by the definition of the semantics the conclusion holds analogously to the cases above.

- **Case** $e = \texttt{"\{"}\ \mathit{lef}\ \texttt{"\}"}$ for $\mathit{lef} \in \texttt{<list-exp-field>}$: Then $T = \{a_1 : T_1, \ldots, a_n : T_n\}$ for given $a_i \in \mathcal{V}, T_i \in \mathcal{T}$ for $i \in \mathbb{N}_0^n$ and $n \in \mathbb{N}$. By the premise of the inference rule, we assume $\Gamma, \Delta \vdash \mathit{lef} : \{a_1 : T_1, \ldots, a_n : T_n\}$ can be derived. By induction hypothesis Theorem 5.7 we can therefore conclude $[\![\mathit{lef}]\!]_{\Gamma,\Delta'} = \{a_1 : T_1, \ldots, a_n : T_n\}$. Thus, by the definition of the semantics the conclusion holds analogously to the cases above.

- **Case** $e = \texttt{"\{\}"}$: Then $T = \{\}$ and $[\![\texttt{"\{\}"}]\!]_{\Gamma,\Delta'} = \{\}$. Thus the conclusion holds.

- **Case** $e = \texttt{"\{"}\ a\ \texttt{"|"}\ \mathit{lef}\ \texttt{"\}"}$ for $a \in \mathcal{V}$ and $\mathit{lef} \in \texttt{<list-exp-field>}$: Then $T = \{a_1 : T_1, \ldots, a_n : T_n, \ldots\}$ for given $a_i \in \mathcal{V}, T_i \in \mathcal{T}$ for $i \in \mathbb{N}_0^n$ and $n \in \mathbb{N}$. By the premise of the inference rule, we assume $(a, \{a_1 : T_1, \ldots, a_n : T_n, \ldots\}) \in \Delta$ and $\Gamma, \Delta \vdash \mathit{lef} : \{a_1 : T_1, \ldots, a_n : T_n\}$ can be derived. By induction hypothesis of Theorem 5.7 we can therefore conclude $[\![\mathit{lef}]\!]_{\Gamma,\Delta'} = \{a_1 : T_1, \ldots, a_n : T_n\}$. The semantic requires that there exists an $e \in \text{value}_\Gamma(\overline{\Gamma}(\{a_1 : T_1, \ldots, a_n : T_n, \ldots\}))$ such that $(a, e) \in \Delta'$. We know Δ' is similar to Δ and therefore this is a valid

assumption. Thus, the semantic is sound and by its definition the conclusion holds analogously to the cases above.

- **Case** $e = a_0$ `"."` a_1 for $a_0, a_1 \in \mathcal{V}$: By the premise of the inference rule we assume $(a_0, \{a_1 : T, \dots\}) \in \Delta$. The semantic requires that there exists an $e \in \text{value}_\Gamma(\overline{\Gamma}(\{a_1 : T, \dots\}))$ such that $(a_0, e) \in \Delta'$. We know Δ' is similar to Δ and therefore this is a valid assumption. Thus, the semantic is sound and by its definition the conclusion holds analogously to the cases above.

- **Case** $e =$ `"let"` *mes* a `"="` e_1 `"in"` e_2 for *mes* $\in$ `<maybe-exp-sign>`, $a \in \mathcal{V}$, $e_1, e_2 \in$ `<exp>`: By the premise of the inference rule we assume $\Gamma, \Delta \vdash e_1 : T_1$ and $\Gamma, \Delta \cup \{(a, \overline{\Gamma}(T_1))\} \vdash e_2 : T_2$ can be derived. Then, by induction hypothesis of Theorem 5.9 we know $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1))$. Therefore, by Theorem 5.4 we know $\Delta' \cup \{(a, [\![e_1]\!]_{\Gamma,\Delta'})\}$ is similar to $\Delta \cup \{(a, \overline{\Gamma}(T_1))\}$. By induction hypothesis of Theorem 5.9 we also know $[\![e_2]\!]_{\Gamma,\Delta' \cup \{(a,[\![e_1]\!]_{\Gamma,\Delta'})\}} \in \text{value}_\Gamma(\overline{\Gamma}(T_2))$ and thus by the definition of the semantics the conclusion holds analogously to the cases above.

- **Case** $e = e_1\ e_2$ for $e_1, e_2 \in$ `<exp>`: By the premise of the inference rule we assume $\Gamma, \Delta \vdash e_1 : T_1 \to T$ and $\Gamma, \Delta \vdash e_2 : T_1$ can be derived. Therefore, by the induction hypothesis of Theorem 5.9 we know, $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1 \to T))$ and $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1))$. Then by the definition of the semantics the conclusion holds analogously to the cases above.

- **Case** $e = b$ for $b \in$ `<bool>`: Then $T = \mathit{Bool}$. By the premise of the inference rule we assume $b : T$ can be derived and by Theorem 5.5 the conclusion holds.

- **Case** $e = i$ for $i \in$ `<int>`: Then $T = \mathit{Int}$. By the premise of the inference rule we assume $b : T$ can be derived and by Theorem 5.6 the conclusion holds.

- **Case** $e =$ `"["` *le* `"]"` for $i \in$ `<list-exp>`: Then $T = \mathit{List}\ T_1$ for $T_1 \in \mathcal{T}$. By the premise of the inference rule we assume $\Gamma, \Delta \vdash \mathit{le} : T$ can be derived and by induction hypothesis Theorem 5.8 the conclusion holds.

- **Case** $e =$ `"("` e_1 `","` e_2 `")"` for $e_1, e_2 \in$ `<exp>`: Then $T = (T_1, T_2)$ for $T_1, T_2 \in \mathcal{T}$. By the premise of the inference rule we assume $\Gamma, \Delta \vdash e_1 : T_1$ and $\Gamma, \Delta \vdash e_2 : T_2$. Therefore, by the induction hypothesis of Theorem 5.9 we know $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1))$ and $[\![e_2]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_2))$. Then by the definition of the semantics the conclusion holds analogously to the cases above.

- **Case** $e =$ `"\"` a `"->"` e for $a \in \mathcal{V}, e \in$ `<exp>`: Then $T = T_1 \to T_2$ for $T_1, T_2 \in \mathcal{T}$. By the premise of the inference rule we assume $\Gamma, \Delta' \cup \{(a, \overline{\Gamma}(T_1))\} \vdash e : T_2$ can be derived. We now need to show that $[\![e]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(T_1 \to T_2)$. We know $[\![e]\!]_{\Gamma,\Delta'} = \lambda b.[\![e]\!]_{\Gamma,\Delta \cup \{(a,b)\}}$ for $b \in \mathcal{V}$. We will therefore by the

definition of the abstraction in the lambda expression let $b \in \text{value}_\Gamma(\overline{\Gamma}(T_1))$ and show $[\![e]\!]_{\Gamma,\Delta' \cup \{(a,b)\}} \in \text{value}_\Gamma(T_2)$. By Theorem 5.4 $\Delta' \cup \{(a,b)\}$ is similar to $\Delta \cup \{(a, \overline{\Gamma}(T_1))\}$ and therefore by induction hypothesis of Theorem 5.9 we conclude $[\![e]\!]_{\Gamma,\Delta' \cup \{(a,b)\}} \in \text{value}_\Gamma(T_2)$. Thus the conclusion holds.

- **Case** $\Gamma, \Delta \vdash c : T$ for $c \in \mathcal{V}$: By the premise of the inference rule we assume $(c, T) \in \Delta$. The semantic requires that there exists an $e \in \text{value}_\Gamma(\overline{\Gamma}(T))$ such that $(c, e) \in \Delta'$. Δ' is similar to Δ and therefore this is a valid assumption. Thus, the semantic is sound and by its definition the conclusion holds analogously to the cases above.

□

3.5.4. Soundness of the Statement Rules

Statements are modelled as operations on either the type context or the variable context. We will now show that the result of the inference rules conforms to their semantics.

Theorem 5.10

Let $lsv \in$ `<list-statement-var>`, $a_i \in \mathbb{N}_1^n$ for $n \in \mathbb{N}_0$. Assume $lsv : (a_1, \ldots, a_n)$ can be derived.

—

Then $[\![lsv]\!] \in \mathcal{V}^*$.

Proof. Let $lsv \in$ `<list-statement-var>`, $a_i \in \mathbb{N}_1^n$ for $n \in \mathbb{N}_0$. Assume $lsv : (a_1, \ldots, a_n)$ can be derived.

- **Case** $lsv =$ `""` and $n = 0$: Then $[\![lsv]\!] = ()$ and thus the conclusion holds.
- **Case** $lsv = a_1\ lsv_1$ for $lsv_1 \in$ `<list-statement-var>`: Then by the inference rule of lsv, we assume that $lsv_1 : (a_2, \ldots, a_n)$ can be derived. Then by induction hypothesis $[\![lsv_1]\!] = (a_2, \ldots, a_n)$, and therefore $[\![lsv]\!] = (a_1, \ldots, a_n)$. Thus the conclusion holds.

□

Theorem 5.11

Let $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$ be type contexts and Δ_1' be a variable context similar to Δ_1 respectively with respect to Γ. Let $s \in$ `<statement>` and assume $\Gamma_1, \Delta_1, s \vdash$

Γ_2, Δ_2 can be derived.

—

Then $[\![s]\!](\Gamma_1, \Delta_1') = (\Gamma_2, \Delta_2')$ for a variable context Δ_2' similar to Δ_2 with respect to Γ.

Proof. Let $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$ be type contexts and Δ_1', Δ_2' be a variable context similar to Δ_1, Δ_2 respectively with respect to Γ_1, Δ_2 respectively. Let $s \in$ `<statement>` and assume $\Gamma_1, \Delta_1, s \vdash \Gamma_2, \Delta_2$ can be derived.

- **Case** $s = mss\ a$ `"="` e for $mss \in$ `<maybe-statement-sort>`, $a \in \mathcal{V}$, $e \in$ `<exp>`, $\Gamma_1 = \Gamma_2$ and $\Delta_2 = \Delta_1 \cup \{(a, \overline{\Gamma_1}(T))\}$ for $T \in \mathcal{T}$: Then from the premise of the inference rule, we assume that $\Gamma_1, mss \vdash e : T$ and $\Gamma_1, \Delta_2 \vdash e : T$ can both be derived. By Theorem 5.9, we know $[\![e]\!]_{\Gamma_1, \Delta_1'} \in \text{value}_{\Gamma_1}(\overline{\Gamma_1}(T))$. Let $\Delta_2' = \Delta_1' \cup \{(a, [\![e]\!]_{\Gamma_1, \Delta_1'})\}$. Then $[\![s]\!](\Gamma_1, \Delta_1') = (\Gamma_2, \Delta_2')$. By Theorem 5.4 Δ_2' is similar to Δ_2.

- **Case** $s =$ `"type alias"` $c\ lsv$ `"="` t for $lsv \in$ `<list-statement-variable>`, $c \in \mathcal{V}$ such that $\Delta_1 = \Delta_2$ and $(c, \_) \notin \Gamma_1$: Let $\Delta_1' = \Delta_2'$. From $[\![s]\!](\Gamma_1, \Delta_1') = (\Gamma_2, \Delta_2')$ the conclusion trivially holds.

□

Theorem 5.12

Let $\Gamma_1, \Delta_1, \Gamma_2, \Delta_2$ be type contexts and Δ_1', be a variable context similar to Δ_1 with respect to Γ. Let $ls \in$ `<list-statement>` such that $\Gamma_1, \Delta_1, ls \vdash \Gamma_2, \Delta_2$ can be derived.

—

Then $[\![ls]\!](\Gamma_1, \Delta_1') = (\Gamma_2, \Delta_2')$ for a variable context Δ_2' similar to Δ_2 with respect to Γ.

Proof. $\Gamma_1, \Delta_1, \Gamma_2, \Delta_2$ be type contexts and Δ_1', be a variable context similar to Δ_1 with respect to Γ. Let $ls \in$ `<list-statement>` such that $\Gamma_1, \Delta_1, ls \vdash \Gamma_2, \Delta_2$ can be derived.

- **Case** $ls =$ `""` for $\Gamma_1 = \Gamma_2$ and $\Delta_1 = \Delta_2$: Let $\Delta_1' = \Delta_2'$. Then $[\![ls]\!] = id$ and therefore the conclusion holds.

- **Case** $ls = s$ `";"` ls_1 for $s \in$ `<statement>` and $ls_1 \in$ `<statement-list>`: From the premise of the inference rule, we assume $\Gamma_1, \Delta_1, s \vdash \Gamma_3, \Delta_3$ and $\Gamma_3, \Delta_3, ls_1 \vdash \Gamma_2, \Delta_2$ for some type contexts Γ_2, Δ_2. We know by Theorem 5.11 that $[\![s]\!](\Gamma_1, \Delta_1') = (\Gamma_3, \Delta_3')$ for a given variable context Δ_3' similar to Δ_3 with

respect to Γ. Also, by induction hypothesis we know $[\![ls_1]\!](\Gamma_3, \Delta'_3) = (\Gamma_2, \Delta'_2)$ for a given Δ'_2 similar to Δ_2 with respect to Γ. Thus $[\![ls]\!] = [\![s]\!] \circ [\![ls_1]\!]$ and therefore the conclusion holds.

□

3.5.5. Soundness of the Program Rules

A program is a sequence of statements. Starting with an empty type context, and an empty variable context, one statement at the time will be applied, resulting in a value e, a type T and a type context Γ such that $e \in \text{value}_\Gamma(T)$.

Theorem 5.13

Let $p \in$ `<program>` and $T \in \mathcal{T}$. Assume $p : T$ can be derived.

—

Then there exist type contexts Γ and Δ such that $[\![p]\!] \in \text{value}_\Gamma(\overline{\Gamma}(T))$.

Proof. Let ls mms `"main="` $e \in$ `<program>`, $ls \in$ `<list-statement>`, $mms \in$ `<maybe-main-sign>` and $e \in$ `<exp>`. Assume $p : T$ for $T \in \mathcal{T}$, $\varnothing, \varnothing, ls \vdash \Gamma, \Delta$ and $\Gamma, \Delta \vdash e : T$ can be derived for type contexts Γ and Δ.

The assumption of Theorem 5.12, namely that $\varnothing, \varnothing, ls \vdash \Gamma, \Delta$ can be derived, holds. By appling said theorem we obtain $[\![ls]\!](\varnothing, \varnothing) = (\Gamma, \Delta')$ for a variable context Δ' similar to Δ with respect to Γ. Therefore, $[\![p]\!] = [\![e]\!]_{\Gamma,\Delta'}$. We know $\Gamma, \Delta \vdash e : T$ and thus by Theorem 5.9 we know that $[\![e]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T))$ and therefore the conclusion holds. □

4. Liquid Types for Elm

In this chapter we will specify a version of liquid types for Elm that can be inferred using an arbitrary SMT solver. Section 4.1 will formally define liquid types. Section 4 will extend the type system of Elm with liquid types. This included the syntax, type inference rules and the denotational semantic. Section 4.3 gives a proof that the extended denotational semantic is sound with respect to the extended type inference rules. Section 4.4 explains how the SMT statements used for the SMT solver can be generated and proves that the provided algorithm is correct.

4.1. Notion of Liquid Types

So-called *refinement types* exclude values from existing types by using a predicate (in this context also called a *refinement*). The definition of such a refinement can be chosen quite freely, but it is important to note that one will also need to provide an algorithm to validate such refinements. This motivates the use of SMT solvers and refinements tailored to the capabilities of specific solvers. Such a set of refinement types are for example *liquid types* (**l**og**i**cally **qu**al**i**fie**d** data types).

We start by defining the syntax and semantic of valid refinements.

Definition 1.1: Logical Qualifier Expressions

We define the set of logical qualifier expressions $\mathcal{Q}$ as follows:

$$
\begin{aligned}
\mathit{IntExp} ::= \; & \mathbb{Z} \\
| \; & \mathit{IntExp} + \mathit{IntExp} \\
| \; & \mathit{IntExp} \cdot \mathbb{Z} \\
| \; & \mathcal{V}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{Q} ::= \; & \mathit{True} \\
| \; & \mathit{False} \\
| \; & \mathit{IntExp} < \mathcal{V} \\
| \; & \mathcal{V} < \mathit{IntExp} \\
| \; & \mathcal{V} = \mathit{IntExp} \\
| \; & \mathcal{Q} \wedge \mathcal{Q} \\
| \; & \mathcal{Q} \vee \mathcal{Q} \\
| \; & \neg \mathcal{Q}
\end{aligned}
$$

Definition 1.2: Well-Formed Logical Qualifier Expressions

Let $e \in \mathcal{Q}$. Let $\Theta : \mathcal{V} \nrightarrow \mathbb{N}$.

—

We say e is *well formed* with respect to Θ iff for all variables v in e, $\Theta(v)$ is well-defined, meaning $\exists n \in \mathbb{N}.(v, n) \in \Theta$.

Definition 1.3: Semantics of Logical Qualifier Expressions

We define the semantic of arithmetic expressions *IntExp* as follows.

$$
\begin{aligned}
[\![.]\!]_{.} : \mathit{IntExp} \to (\mathcal{V} \nrightarrow \mathbb{N}) \to \mathbb{N} \\
[\![n]\!]_{\Theta} =& n \\
[\![i + j]\!]_{\Theta} =& [\![i]\!]_{\Theta} + [\![j]\!]_{\Theta} \\
[\![i \cdot n]\!]_{\Theta} =& [\![i]\!]_{\Theta} \cdot n \\
[\![a]\!]_{\Theta} =& \Theta(a)
\end{aligned}
$$

Note that we assume that the given expression is well-formed with respect to Θ.

We also define the semantic of logical qualifier expressions $\mathcal{Q}$ as follows:

$$
\begin{aligned}
[\![.]\!]_{.} : \mathcal{Q} \to (\mathcal{V} \nrightarrow \mathbb{N}) \to \mathit{Bool} \\
[\![\mathit{True}]\!]_\Theta &= \mathit{True} \\
[\![\mathit{False}]\!]_\Theta &= \mathit{False} \\
[\![i < a]\!]_\Theta &= ([\![i]\!]_\Theta < [\![a]\!]_\Theta) \\
[\![a < i]\!]_\Theta &= ([\![a]\!]_\Theta < [\![i]\!]_\Theta) \\
[\![a = i]\!]_\Theta &= ([\![a]\!]_\Theta = [\![i]\!]_\Theta) \\
[\![p \wedge q]\!]_\Theta &= ([\![p]\!]_\Theta \wedge [\![q]\!]_\Theta) \\
[\![p \vee q]\!]_\Theta &= ([\![p]\!]_\Theta \vee [\![q]\!]_\Theta) \\
[\![\neg p]\!]_\Theta &= (\neg [\![p]\!]_\Theta)
\end{aligned}
$$

We will now extend our previous definition of types (see Definition 1.1) by the notion of refinement types. This extension is not very interesting, as refinement types don't behave differently from their underlying type.

Definition 1.4: Extended Types

We define the following

T is a *mono type* $:\Leftrightarrow$

$$
\begin{aligned}
& T \text{ is a type variable} \\
\vee\ & T \text{ is a type application} \\
\vee\ & T \text{ is a algebraic type} \\
\vee\ & T \text{ is a product type} \\
\vee\ & T \text{ is a function type} \\
\vee\ & T \text{ is a liquid type}
\end{aligned}
$$

T is a *poly type* $:\Leftrightarrow$

$$
\begin{aligned}
& T = \forall a.T' \\
& \text{where } T' \text{ is a mono type} \\
& \text{or poly type and } a \text{ is a symbol.}
\end{aligned}
$$

T is a *type*$:\Leftrightarrow$

$$
\begin{aligned}
& T \text{ is a mono type} \\
\vee\ & T \text{ is a poly type.}
\end{aligned}
$$

by using the predicates:

$$
\begin{aligned}
T \text{ is a } \textit{type variable} &:\Leftrightarrow T \in \mathcal{V} \\
T \text{ is a } \textit{type application} &:\Leftrightarrow T \text{ is of form } C\ T_1 \ldots T_n \\
&\quad \text{where } n \in \mathbb{N}, C \in \mathcal{V} \text{ and the } T_i \text{ are mono types for} \\
&\quad \text{all } i \in \mathbb{N}_1^n. \\
T \text{ is a } \textit{algebraic type} &:\Leftrightarrow T \text{ is of form} \\
&\quad \mu C.C_1\ T_{1,1} \ldots T_{1,k_1} \mid \ldots \mid C_n\ T_{n,1} \ldots T_{n,k_n} \\
&\quad \text{such that } \exists i \in \mathbb{N}.\forall j \in \mathbb{N}_1^{k_i}.T_{i,j} \neq C \\
&\quad \text{where } n \in \mathbb{N}, C \in \mathcal{V}, k_i \in \mathbb{N}_0 \text{ for all } i \in \mathbb{N}_1^n \\
&\quad \text{and } \ T_{i,k_i} \text{ is a mono type or } T_{i,k_i} = C \text{ for all } i \in \mathbb{N}_1^n \\
&\quad \text{and } j \in \mathbb{N}_1^{k_i}. \\
T \text{ is a } \textit{product type} &:\Leftrightarrow T \text{ is of form } \{l_1 : T_1, \ldots, l_n : T_n\} \\
&\quad \text{where } n \in \mathbb{N}_0 \text{ and } l_i \in \mathcal{V} \text{ and } T_i \text{ are mono types for} \\
&\quad \text{all } i \in \mathbb{N}_1^n. \\
T \text{ is a } \textit{function type} &:\Leftrightarrow T \text{ is of form } T_1 \to T_2 \\
&\quad \text{where } T_1 \text{ and } T_2 \text{ are mono types.} \\
T \text{ is a } \textit{liquid type} &:\Leftrightarrow T \text{ is of form } \{a : \mathit{Int} \mid r\} \\
&\quad \text{where } \ a \in \mathcal{V}, r \in \mathcal{Q}, \mathit{Nat} := \mu C.1 \mid \mathit{Succ}\ C \\
&\quad \text{and } \mathit{Int} := \mu\_.0 \mid \mathit{Pos}\ \mathit{Nat} \mid \mathit{Neg}\ \mathit{Nat}. \\
&\vee\ T \text{ is of form } a : \{b : \mathit{Int} \mid r\} \to T_0 \\
&\quad \text{where } a, b \in \mathcal{V}, r \in \mathcal{Q}, \text{ and } T_0 \ \text{ is a liquid type.}
\end{aligned}
$$

We will also need to redefine the definition of free variables and type substitution. The only change is the trival addition of refinement types.

Definition 1.5: Bound, Free, Set of free variables

Let a be a type variable and T be a type

—

We say

- a is *free* in $T :\Leftrightarrow a \in \mathit{free}(T)$
- a is *bound* in $T :\Leftrightarrow a \notin \mathit{free}(T)$ and a occurs in T.

where

$$
\begin{aligned}
\text{free}(a) :=& \{a\} \\
\text{free}(C\ T_1 \dots T_n) :=& \bigcup_{i \in \mathbb{N}_1^n} \text{free}(T_i) \\
\text{free}\begin{pmatrix} \mu C. \\ C_1\ T_{1,1} \dots\ T_{1,k(1)} \\ | \dots \\ |\ C_n\ T_{n,1} \dots\ T_{n,k(n)} \end{pmatrix} :=& \bigcup_{i \in \mathbb{N}_0^n} \bigcup_{j \in \mathbb{N}_0^{k_i}} \begin{cases} \varnothing & \text{if } T_{i,j} = C \\ \text{free}(T_{i,j}) & \text{else} \end{cases} \\
\text{free}(\{\_ : T_1, \dots, \_ : T_n\}) :=& \bigcup_{i \in \mathbb{N}_1^n} \text{free}(T_i) \\
\text{free}(T_1 \to T_2) :=& \text{free}(T_1) \cup \text{free}(T_2) \\
\text{free}(\forall a.T) :=& \text{free}(T) \backslash \{a\} \\
\text{free}(\{a : \mathit{Int} \mid r\}) :=& \{\} \\
\text{free}(a : \{b : \mathit{Int} \mid r\} \to T) :=& \{\}
\end{aligned}
$$

We will now redefine the notion of values. As mentioned before, liquid types exclude values that do not ensure a specific refinement.

Definition 1.6: Values

Let $\mathcal{S}$ be the class of all finite sets and Γ be a type context.

—

We define

$$
\begin{aligned}
\text{values}_\Gamma &: \mathcal{V} \to \mathcal{S} \\
\text{values}_\Gamma(a) &:= \text{values}_\Gamma(\Gamma(a)) \\
\text{values}_\Gamma(C\ T_1\ \ldots\ T_n) &:= \text{values}_\Gamma(\overline{\Gamma(C)}(T_1, \ldots, T_n)) \\
\text{values}_\Gamma \begin{pmatrix} \mu C. \\ \mid C_1\ T_{1,1} \ldots\ T_{1,k_1} \\ \mid \ldots \\ \mid C_n\ T_{n,1} \ldots\ T_{n,k_n} \end{pmatrix} &:= \bigcup_{i \in \mathbb{N}_0} \text{rvalues}_\Gamma \begin{pmatrix} & \mu C. \\ i, & \mid C_1\ T_{1,1} \ldots\ T_{1,k_1} \\ & \mid \ldots \\ & \mid C_n\ T_{n,1} \ldots\ T_{n,k_n} \end{pmatrix} \\
\text{values}_\Gamma(\{l_1 : T_1, \ldots, l_n : T_n\}) &:= \Big\{ \{l_1 = t_1, \ldots, l_n = t_n\} \\
&\qquad \mid \forall i \in \mathbb{N}_1^n . t_i \in \text{values}_\Gamma(T_i) \Big\} \\
\text{values}_\Gamma(T_1 \to T_2) &:= \{ f \mid f : \text{values}_\Gamma(T_1) \to \text{values}_\Gamma(T_2) \} \\
\text{values}_\Gamma(\forall a.T) &:= \lambda b.\text{values}_{\{(a,b)\} \cup \Gamma}(T) \text{ where the symbol } b \text{ does} \\
&\qquad \text{not occur in } T. \\
\text{values}_\Gamma(\{a : \mathit{Int} \mid r\}) &:= \text{refinedValues}_{\{\}}(\{a : \mathit{Int} \mid r\}) \\
\text{values}_\Gamma(a : \{b : \mathit{Int} \mid r\} \to T) &:= \text{refinedValues}_{\{\}}(a : \{b : \mathit{Int} \mid r\} \to T)
\end{aligned}
$$

using the following helper functions.

Let $l \in \mathbb{N}, T := \mu C. \mid C_1\ T_{1,1} \ldots\ T_{1,k(1)} \mid \ldots \mid C_n\ T_{n,1} \ldots\ T_{n,k(n)}$. We define:

$$
\text{rvalues}_\Gamma(0, T) := \left\{ C_i\ v_1 \ldots v_n \;\middle|\; \begin{array}{l} i \in \mathbb{N}_1^n \\ \wedge \forall j \in \mathbb{N}_1^{k(i)} . T_{i,j} \neq C \wedge v_j \in \text{values}_\Gamma(T_{i,j}) \end{array} \right\}
$$

$$
\text{rvalues}_\Gamma(l+1, T) := \left\{ C_i\ v_1 \ldots v_n \;\middle|\; \begin{array}{l} i \in \mathbb{N}_1^n \\ \wedge \forall j \in \mathbb{N}_1^{k(i)} . v_j \in \begin{cases} \text{rvalues}_\Gamma(l, T) & \text{if } T_{i,j} = C \\ \text{values}_\Gamma(T_{i,j}) & \text{else} \end{cases} \end{array} \right\}
$$

Let $\Theta : \mathcal{V} \nrightarrow \mathbb{N}$. We define:

$$
\begin{aligned}
&\text{refinedValues}_\Theta(\{a : \mathit{Int} \mid r\}) := \\
&\qquad \{ n \in \text{values}_{\{\}}(\mathit{Int}) \\
&\qquad \mid r \text{ is well formed with respect to } \Theta \cup \{(a,n)\} \wedge\ [\![r]\!]_{\Theta \cup \{(a,n)\}} \}
\end{aligned}
$$

$$
\begin{aligned}
&\text{refinedValues}_\Theta(a : \{b : \mathit{Int} \mid r\} \to T) := \\
&\qquad \{ b \in \text{refinedValues}_\Theta(\{b : \mathit{Int} \mid r\} \to T) \\
&\qquad \mid \forall n \in \text{refinedValues}_\Theta(\{b : \mathit{Int} \mid r\}). \\
&\qquad\quad b(n) \in \text{refinedValues}_{\Theta \cup \{(a,n)\}}(T) \}
\end{aligned}
$$

4.2. Liquid Types for Elm

We will now extend the type system of Elm with liquid types.

4.2.1. Syntax

We will use the syntax described in the Section 4.1.

Definition 2.1: Extended Type Signature Syntax

Let `<upper-var>` and `<lower-var>` be two variable domains.

—

We define the following syntax:

```
<int-exp-type> ::=<int>
                 | <int-exp-type> + <int-exp-type>
                 | <int-exp-type> * <int>
                 | 𝒱

<qualifier-type> ::="True""
                   | "False"
                   | "(<)" <int-exp-type> v
                   | "(<)" v <int-exp-type>
                   | "(==)" v <int-exp-type>
                   | "(&&)" <qualifier-type> <qualifier-type>
                   | "(||)" <qualifier-type> <qualifier-type>
                   | "not" <qualifier-type>

<liquid-type> ::=
      "{v:Int|" <qualifier-type> "}"
      | <lower-var> ":{v:Int|" <qualifier-type> "->" <liquid-type>
```

```
<type> ::=<liquid-type>
        | "Bool"
        | "List" <type>
        | "(" <type> "," <type> ")"
        | "{" <list-type-fields> "}"
        | <type> "->" <type>
        | <upper-var> <list-type>
        | <lower-var>
```

4.2.2. Type Inference

We will also extend the inference rules. The interesting part is the new judgment for `<exp>`: We introduce two new sets Θ and Λ. As before, Θ will contain the type of a variable. This is similar to Section 4.1 where Θ contained the value of a variable. The set Λ contains boolean expressions that get collected while traversing if-branches. We will use these expressions to allow path sensitive subtyping.

4.2.2.1. Type Signature Judgments

For type signature judgments, let $exp \in IntExp$, $q \in \mathcal{Q}$. Let Γ, Δ be type contexts. Let $\Lambda \subset \mathcal{Q}$ and $\Theta : \mathcal{V} \nrightarrow \mathcal{Q}$.

For $iet \in$ `<int-exp-type>`, the judgment has the form

$$iet : exp$$

which can be read as "iet corresponds to exp".

For $qt \in$ `<qualifier-type>`, the judgment has the form

$$qt : q$$

which can be read as "qt corresponds to q"

For $lt \in$ `<liquid-type>`, the judgment has the form

$$lt :_{\Theta} T$$

which can be read as "lt corresponds to the liquid type T with respect to Θ".

As previously already stated, for $t \in$ `<type>` the judgment has the form

$$\Gamma \vdash t : T$$

which can be read as "given Γ, t has the type T".

For $e \in$ `<exp>` the judgment has the form

$$\Gamma, \Delta, \Theta, \Lambda \vdash e : T$$

which can be read as "given Γ, Δ, Θ and Λ, e has the type T".

4.2.3. Auxiliary Definitions

We will use all auxiliary definitions defined in Section 3.3.

Well-formed Liquid Type

We have already defined well-formed logical qualifiers expressions. We will now extend the notion to well-formed liquid types.

Definition 2.2: Well-formed Liquid Type

Let $\Theta : \mathcal{V} \nrightarrow \mathcal{T}$.

—

We define following.

$$\begin{aligned}
&\text{wellFormed} \subseteq \{t \in \mathcal{T} \mid t \text{ is a liquid type}\} \times (\mathcal{V} \nrightarrow \mathbb{N}) \\
&\text{wellFormed}(\{b : \mathit{Int} \mid r\}, \{(a_1, r_1), \ldots, (a_n, r_n)\}) :\Leftrightarrow \\
&\qquad \forall k_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r_1\}) \ldots \forall k_n \in \text{value}_\Gamma(\{nu : \mathit{Int} \mid r_n\}). \\
&\qquad r \text{ is well defined with respect to } \{(a_1, k_1), \ldots, (a_n, k_n), (b, \mathit{Int})\} \\
&\text{wellFormed}(a : \{b : \mathit{Int} \mid r\} \to T, \Theta) :\Leftrightarrow \\
&\qquad (a, _) \notin \Theta \\
&\quad \wedge \text{wellFormed}(\{b : \mathit{Int} \mid r\}, \Theta) \\
&\quad \wedge \text{wellFormed}(T, \Theta \cup \{(a, \{b : \mathit{Int} \mid r\})\})
\end{aligned}$$

Subtyping

There are some liquid types whos values are a subset of the values from another type. In this case we say it is a subtype. For our use case we will use a different definition of subtyping.

Definition 2.3: Subtyping

Let $\Theta : \mathcal{V} \nrightarrow \mathcal{T}$. Let $\Lambda \subset \mathcal{Q}$, $r_1, r_2 \in \mathcal{Q}$

—

We define the following.

$$
\begin{aligned}
\{a_1 : \mathit{Int}|q_1\} <:_{\Theta,\Lambda} \{a_2 : \mathit{Int}|q_2\} \ :\Leftrightarrow \\
&\text{Let } \{(b_1, r_1), \ldots, (b_n, r_n)\} = \Theta \text{ in} \\
&\forall k_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int}|r_1\}). \ldots \forall k_n \in \text{value}_\Gamma(\{\nu : \mathit{Int}|r_n\}). \\
&\forall n \in \mathbb{Z}. \forall e \in \Lambda. \\
&\quad [\![e]\!]_{\{(a_1,n),(b_1,k_1),\ldots,(b_n,k_n)\}} \\
&\quad \wedge [\![q_1]\!]_{\{(a_1,n),(b_1,k_1),\ldots,(b_n,k_n)\}} \\
&\Rightarrow [\![q_2]\!]_{\{(a_2,n),(b_1,k_1),\ldots,(b_n,k_n)\}} \\
a_1 : \{b_1 : \mathit{Int}|q_1\} \to T_2 <:_{\Theta,\Lambda} a_1 : \{b_2 : \mathit{Int}|q_2\} \to T_4 \ :\Leftrightarrow \\
&\{b_2 : \mathit{Int}|q_2\} <:_{\Theta,\Lambda} \{b_1 : \mathit{Int}|q_1\} \\
&\wedge T_2 <:_{\Theta \cup \{(a_1, \{b_2 : \mathit{Int}|r_2\})\},\Lambda} T_4
\end{aligned}
$$

For two liquid types T_1, T_2, we say T_1 is a subtype of T_2 with respect to Θ and Λ if and only if $T_1 <:_{\Theta,\Lambda} T_2$ is valid.

Subtyping comes with an additional inference rule for `<exp>`. The sharpness of the inferred subtype depends on the capabilities of the SMT-Solver. Using this optional inference rule, the SMT-Solver will need to find the sharpest subtype, or at least sharp enough: In the case of type checking, it might be that the subtype is too sharp and therefore the SMT-Solver can't check the type successfully.

$$
\frac{\Gamma, \Delta, \Theta, \Lambda \vdash e : T_1 \quad T_1 <:_{\Theta,\Lambda} T_2 \quad \text{wellFormed}(T_2, \Theta)}{\Gamma, \Delta, \Theta, \Lambda \vdash e : T_2}
$$

We will discuss in Chapter 4.4, how one can use this inference rule for automated type checking.

4.2.4. Inference Rules for Type Signatures

Inference Rules for `<int-exp-type>`

Judgment: $iet : exp$

$$\frac{i : Int}{i : i}$$

$$\frac{iet_1 : exp_1 \quad iet_2 : exp_2 \quad exp_1 + exp_2 = exp_3}{iet_1 \texttt{ + } iet_2 : exp_3}$$

$$\frac{i : Int \quad iet : exp_0 \quad exp_0 * i = exp_1}{iet \texttt{ * } i : exp_1}$$

$$\frac{a = exp}{a : exp}$$

Inference Rules for `<qualifier-type>`

Judgment: $qt : q$

This judgment is used to convert from `<qualifier-type>` to $\mathcal{Q}$.

$$\frac{}{\texttt{True} : True}$$

$$\frac{}{\texttt{False} : False}$$

$$\frac{iet : exp_0 \quad exp_0 < \nu = q}{\texttt{(<)} \ iet \ \texttt{v} : q}$$

Note that we replace the letter `v` with a special character ν.

$$\frac{iet : exp_0 \quad \nu < exp_0 = q}{\texttt{(<)} \ \texttt{v} \ iet : q}$$

$$\frac{iet : exp_0 \quad (\nu = exp_0) = q}{\texttt{(=)} \ \texttt{v} \ iet : q}$$

$$\frac{qt_1 : q_1 \quad qt_2 : q_2 \quad q_1 \wedge q_2 = q_3}{\texttt{(\&\&)} \ qt_1 \ qt_2 : q_3}$$

$$\frac{qt_1 : q_1 \quad qt_2 : q_2 \quad q_1 \vee q_2 = q_3}{\texttt{(||)}\ qt_1\ qt_2 : q_3}$$

$$\frac{qt : q_1 \quad \neg q_1 = q_2}{\texttt{not}\ qt : q_2}$$

Inference Rules for `<liquid-type>`

Judgment: $lt :_{\Theta} T$

$$\frac{qt : q \quad \{\nu : Int|\ q\ \} = T \quad \text{wellFormed}(T_2, \Theta \cup \{(\nu, True)\})}{\texttt{"\{v:Int|"}\ qt\ \texttt{"\}"} :_{\Theta} T}$$

$$\frac{\texttt{"\{v:Int|"}\ qt\ \texttt{"\}"} :_{\Theta} \{\nu : Int|\ q\ \} \quad lt :_{\Theta \cup \{(a,q)\}} T_2 \quad (a : \{\nu : Int|\ q\ \} \rightarrow T_2) = T_3}{a\ \texttt{":"}\ \texttt{"\{v:Int|"}\ qt\ \texttt{"\}"}\ \texttt{"->"}\ lt :_{\Theta} T_3}$$

Inference Rules for `<type>`

Judgment: $\Gamma \vdash t : T$

$$\frac{lt :_{\{\}} T}{\Gamma \vdash lt : T}$$

All other inference rules for types have already been described.

4.2.5. Inference Rules for Expressions

Inference Rules for `<Exp>`

The following are special inference rules for liquid types. For non-liquid types the old rules still apply.

$$\Gamma, \Delta, \Theta, \Lambda \vdash \texttt{"(+)"} : (a : Int \rightarrow b : Int \rightarrow \{\nu : Int \mid \nu = a + b\})$$

$$\Gamma, \Delta, \Theta, \Lambda \vdash \texttt{"(-)"} : (a : Int \rightarrow b : Int \rightarrow \{\nu : Int \mid \nu = a + (-b)\})$$

$$\Gamma, \Delta, \Theta, \Lambda \vdash \texttt{"(*)"} : (a : Int \rightarrow b : Int \rightarrow \{\nu : Int \mid \nu = a * b\})$$

$$\Gamma, \Delta, \Theta, \Lambda \vdash \texttt{"(//)"} : \mathit{Int} \to \{\nu : \mathit{Int} \mid \neg(\nu = 0)\} \to \mathit{Int}$$

Thus, using a liquid type we can avoid dividing by zero.

$$\frac{\Gamma, \Delta, \Theta, \Lambda \vdash e_1 : \mathit{Bool} \quad e_1 : e_1' \qquad \Gamma, \Delta, \Theta, \Lambda \cup \{e_1'\} \vdash e_2 : T \quad \Gamma, \Delta, \Theta, \Lambda \cup \{\neg e_1'\} \vdash e_3 : T}{\Gamma, \Delta, \Theta, \Lambda \vdash \texttt{"if"}\ e_1\ \texttt{"then"}\ e_2\ \texttt{"else"}\ e_3 : T}$$

We add the condition e_1 to Λ and ensure that the resulting liquid type is well-formed. Note that we assume that $e_1 \in$ `<qualifier-type>`. If this is not the case, then the inference rule can not be applied and therefore the judgment can not be derived. In some cases we can recover by falling back to the old rule for non-liquid types, but recovery is not guaranteed.

$$\frac{\Gamma, \Delta, \Theta, \Lambda \vdash e_1 : (a : T_1 \to T_2) \qquad \Gamma, \Delta, \Theta, \Lambda \vdash e_2 : T_1 \quad e_2 : e_2' \quad [T_2]_{\{(a, e_2')\}} = T_3}{\Gamma, \Delta, \Theta, \Lambda \vdash e_1\ e_2 : T_3}$$

We change the type of e_1 to $a : T_1 \to T_2$. To ensure that a can't escape the scope, we substitute it with e_2'. Note that we assume that $e_2 \in$ `<qualifier-type>`, else we can try to recover by using the inference rules for non-liquid types.

$$\frac{a : \{\nu : \mathit{Int} \mid q\} \to T_1 = T_2 \qquad \Gamma, \Delta \cup \{(a, \{\nu : \mathit{Int} \mid q\})\}, \Theta \cup \{(a, q)\}, \Lambda \vdash e : T_1}{\Gamma, \Delta, \Theta, \Lambda \vdash \texttt{"\textbackslash"}\ a\ \texttt{"->"}\ e : T_2}$$

We define the type as $a : T_1 \to T_2 = T_3$. Note that the variable a in the expression realm and the variable a within the context of liquid types are the same. This is because we assume that renaming can be applied at any step of the type inference. To avoid having double bound variables, we require that $a : T_1 \to T_2$ is well-formed.

$$\frac{(a, \{\nu : \mathit{Int} \mid \ q\}) \in \Delta \quad (a, q) \in \Theta}{\Gamma, \Delta, \Theta, \Lambda \vdash a : \{\nu : \mathit{Int} \mid \nu = a\}}$$

We can give a variable a sharp liquid type.

—

All other inference rules for expressions have not changed.

4.2.6. Denotational Semantic

For the denotational semantic we only need to extend the semantic for type signatures.

Definition 2.4: Type Signature Semantic

Let Let Γ be a type context. Let $\Theta : \mathcal{V} \nrightarrow \mathcal{Q}$.

—

$$
\begin{aligned}
[\![.]\!] :& \texttt{<int-exp-type>} \to \mathit{IntExp} \\
[\![n]\!] =& n \\
[\![iet_1 \texttt{ + } iet_2]\!] =& i_1 + i_2 \\
& \text{such that } i_1 = [\![iet_1]\!] \text{ and } i_2 = [\![iet_2]\!] \text{ where } i_1, i_2 \in \mathit{IntExp} \\
[\![iet \texttt{ * } n]\!] =& i \cdot n \\
& \text{such that } i = [\![iet]\!] \text{ where } i \in \mathit{IntExp} \\
[\![a]\!] =& a
\end{aligned}
$$

$$
\begin{aligned}
[\![.]\!] :& \texttt{<qualifier-type>} \to \mathcal{Q} \\
[\![\texttt{"True"}]\!] =& \mathit{True} \\
[\![\texttt{"False"}]\!] =& \mathit{False} \\
[\![\texttt{"(<)"} \ iet \ v]\!] =& i < \nu \\
& \text{such that } i = [\![iet]\!] \text{ where } i \in \mathit{IntExp} \\
[\![\texttt{"(<)"} \ v \ iet]\!] =& \nu < i \\
& \text{such that } i = [\![iet]\!] \text{ where } i \in \mathit{IntExp} \\
[\![\texttt{"(==)"} \ v \ iet]\!] =& \nu = i \\
& \text{such that } i = [\![iet]\!] \text{ where } i \in \mathit{IntExp} \\
[\![\texttt{"(\&\&)"} \ qt_1 \ qt_2]\!] =& q_1 \wedge q_2 \\
& \text{such that } q_1 = [\![qt_1]\!] \text{ and } q_2 = [\![qt_2]\!] \\
& \text{where } q_1 \in \mathcal{Q} \text{ and } q_2 \in \mathcal{Q} \\
[\![\texttt{"(||)"} \ qt_1 \ qt_2]\!] =& q_1 \vee q_2 \\
& \text{such that } q_1 = [\![qt_1]\!] \text{ and } q_2 = [\![qt_2]\!] \\
& \text{where } q_1 \in \mathcal{Q} \text{ and } q_2 \in \mathcal{Q} \\
[\![\texttt{"not"} \ qt]\!] =& \neg q \\
& \text{such that } q = [\![qt]\!] \text{ where } q \in \mathcal{Q}
\end{aligned}
$$

$$
\begin{aligned}
[\![.]\!] : & \texttt{<liquid-type>} \to \mathcal{T} \\
[\![\texttt{"\{v:Int|"}\ qt\ \texttt{"\}"}]\!] = & \{\nu : Int|\ r\ \} \\
& \text{such that } r = [\![qt]\!] \text{ where } r \in \mathcal{Q} \\
[\![a\ \texttt{":"}\ \texttt{"\{v:Int|"}\ qt\ \texttt{"\}"}\ \texttt{"->"}\ lt]\!] = & a : T_1 \to T_2 \\
& \text{such that } T_1 = [\![\texttt{"\{v:Int|"}\ qt\ \texttt{"\}"}]\!], \\
& \text{and } T_2[\![lt]\!] \text{ where } T_1, T_2 \text{ are liquid types}
\end{aligned}
$$

We extend $[\![.]\!]_\Gamma : \texttt{<type>} \to \mathcal{T}$ by $[\![lt]\!]_\Gamma = [\![lt]\!]$ to now allow liquid types as type signatures.

4.3. Soundness of Liquid Types

We will now show that the extension is sound. To do so we first will show the soundness of the inference rules with respect to the new semantics.

Theorem 3.1

Let $iet \in \texttt{<int-exp-type>}$ and $exp \in IntExp$. Assume $iet : exp$ can be derived.

—

Then $[\![iet]\!] = exp$.

Proof. Let $iet \in \texttt{<int-exp-type>}$ and $exp \in IntExp$. Assume $iet : exp$ can be derived.

- **Case** $iet = i$ for $i \in Int$: Then $[\![iet]\!] = i$ and therefore the conclusion holds.
- **Case** $iet = iet_1\ \texttt{+}\ iet_2$ for $iet_1, iet_2 \in \texttt{<int-exp-type>}$: From the premise of the inference rule, we assume that $iet_1 : exp_1$ and $iet_2 : exp_2$ hold. By induction hypothesis $[\![iet_1]\!] = exp_1$ and $[\![iet_2]\!] = exp_2$. Thus $[\![iet]\!] = exp_1 + exp_2$ and therefore the conclusion holds.
- **Case** $iet = iet_1\ \texttt{*}\ i$ for $iet_1 \in \texttt{<int-exp-type>}$ and $i \in Int$: From the premise of the inference rule, we assume that $iet_1 : exp_1$ holds. By induction hypothesis $[\![iet_1]\!] = exp_1$. Thus $[\![iet]\!] = exp_1 \cdot i$ and therefore the conclusion holds.
- **Case** $iet = a$ for $a \in \mathcal{V}$: Then $[\![a]\!] = a$ and therefore the conclusion holds.

□

Theorem 3.2

Let $qt \in$ `<qualifier-type>` and $q \in \mathcal{Q}$. Assume $qt : q$ can be derived.

—

Then $[\![qt]\!] = q$.

Proof. Let $qt \in$ `<qualifier-type>` and $q \in \mathcal{Q}$. Assume $qt : q$ can be derived.

- **Case** $qt =$ `True`: Then $[\![qt]\!] = \mathit{True}$ and therefore the conclusion holds.
- **Case** $qt =$ `False`: Then $[\![qt]\!] = \mathit{False}$ and therefore the conclusion holds.
- **Case** $qt =$ `(<)` iet `v`: From the premise of the inference rule, we assume that $\mathit{iet} : \mathit{exp}$. By Theorem 3.1 $[\![\mathit{iet}]\!] = \mathit{exp}$ for $\mathit{exp} \in \mathit{IntExp}$. Then $[\![qt]\!] = \mathit{exp} < \nu$ and therefore the conclusion holds.
- **Case** $qt =$ `(<)` `v` iet: From the premise of the inference rule, we assume that $\mathit{iet} : \mathit{exp}$. By Theorem 3.1 $[\![\mathit{iet}]\!] = \mathit{exp}$ for $\mathit{exp} \in \mathit{IntExp}$. Then $[\![qt]\!] = \nu < \mathit{exp}$ and therefore the conclusion holds.
- **Case** $qt =$ `(=)` `v` iet: From the premise of the inference rule, we assume that $\mathit{iet} : \mathit{exp}$. By Theorem 3.1 $[\![\mathit{iet}]\!] = \mathit{exp}$ for $\mathit{exp} \in \mathit{IntExp}$. Then $[\![qt]\!] = (\nu = \mathit{exp})$ and therefore the conclusion holds.
- **Case** $qt =$ `(&&)` qt_1 qt_2 for $qt_1, qt_2 \in$ `<qualifier-type>`: From the premise of the inference rule, we assume that $qt_1 : q_1$ and $qt_2 : q_2$ hold for $q_1, q_2 \in \mathcal{Q}$. By induction hypothesis $[\![qt_1]\!] = q_1$ and $[\![qt_2]\!] = q_2$. Thus $[\![qt]\!] = q_1 \wedge q_2$ and therefore the conclusion holds.
- **Case** $qt =$ `(||)` qt_1 qt_2 for $qt_1, qt_2 \in$ `<qualifier-type>`: From the premise of the inference rule, we assume that $qt_1 : q_1$ and $qt_2 : q_2$ hold for $q_1, q_2 \in \mathcal{Q}$. By induction hypothesis $[\![qt_1]\!] = q_1$ and $[\![qt_2]\!] = q_2$. Thus $[\![qt]\!] = q_1 \vee q_2$ and therefore the conclusion holds.
- **Case** $qt =$ `not` qt_1 for $qt_1 \in$ `<qualifier-type>`: From the premise of the inference rule, we assume that $qt_1 : q_1$ holds for $q_1 \in \mathcal{Q}$. By induction hypothesis $[\![qt_1]\!] = q_1$. Thus $[\![qt]\!] = \neg\, q_1$ and therefore the conclusion holds.

□

Theorem 3.3

Let $\Theta : \mathcal{V} \nrightarrow \mathcal{Q}$. Let $lt \in$ `<liquid-type>` and $T \in \mathcal{T}$. Assume $lt :_\Theta T$ can be derived.

—

Then $[\![lt]\!] = T$.

Proof. Let $\Theta : \mathcal{V} \nrightarrow \mathcal{Q}$. Let $lt \in$ `<liquid-type>` and $T \in \mathcal{T}$. Assume $lt :_\Theta T$ can be derived.

- **Case** $lt =$ `"{v:Int|"` qt `"}"` for $qt \in$ `<qualifier-type>`: From the premise of the inference rule, we assume that $qt : q$ for $q \in \mathcal{Q}$ holds. By Theorem 3.2 $[\![qt]\!] = q$. Then $[\![lt]\!] = \{\nu : Int \mid q\}$ and therefore the conclusion holds.
- **Case** $lt = a$ `": {v:Int|"` qt `"} ->"` lt_2 for $a \in \mathcal{V}$, $qt \in$ `<qualifier-type>` and $lt \in$ `<liquid-type>`: From the premise of the inference rule, we assume that `"{v:Int|"` qt `"}"` $:_\Theta \{\nu : Int \mid r_1\}$ and $lt_2 :_{\Theta \cup \{(a, r_1)\}} T_2$ for liquid type T_2 and $r_1 \in \mathcal{Q}$. By induction hypothesis $[\![lt_2]\!] = T_2$. Then $[\![lt]\!] = a : \{\nu : Int \mid r_1\} \to T_2$ and therefore the conclusion holds.

□

We can now again prove the soundness of the semantic of type annotations.

Theorem 3.4

Let Γ be a type context, $t \in$ `<type>` and $T \in \mathcal{T}$. Assume $\Gamma \vdash t : T$ can be derived.

—

Then $[\![t]\!]_\Gamma = T$.

Proof. Let Γ be a type context, $t \in$ `<type>` and $T \in \mathcal{T}$. Assume $\Gamma \vdash t : T$ can be derived.

- **Case** $t = lt$ for $lt \in$ `<liquid-type>`: From the premise of the inference rule, we assume that $lt :_\Theta T$ for liquid type T holds. By Theorem 3.3 $[\![lt]\!] = T$. Then $[\![t]\!] = T$ and therefore the conclusion holds.

All other cases have been proven in Theorem 5.3. □

To finish of, we need to prove the soundness of the inference rules for expressions. This is by the definition of refinement types trivially true, as the set values of a refinement type is always a subtype of the set of values of the base type.

Theorem 3.5

Let Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ. Let $\Lambda \subset \mathcal{Q}$ and $\Theta : \mathcal{V} \nrightarrow \mathcal{Q}$. Let $e \in$ `<exp>` and $T \in \mathcal{T}$. Assume $\Gamma, \Delta, \Theta, \Lambda \vdash e : T$ can be derived.

—

Then $[\![e]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T))$.

Proof. Let Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ. Let $\Lambda \subset \mathcal{Q}$ and $\Theta : \mathcal{V} \nrightarrow \mathcal{Q}$. Let $e \in$ `<exp>` and $T \in \mathcal{T}$. Assume $\Gamma, \Delta, \Theta, \Lambda \vdash e : T$ can be derived.

- **Case** $e =$ `"(+)"`: Then $T = a : \mathit{Int} \to b : \mathit{Int} \to \{\nu : \mathit{Int} \mid \nu = a + b\}$ and $[\![\texttt{"(+)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.n + m$ where $n, m \in \mathbb{Z}$ and thus the conclusion holds.
- **Case** $e =$ `"(-)"`: Then $T = a : \mathit{Int} \to b : \mathit{Int} \to \{\nu : \mathit{Int} \mid \nu = a + (-b)\}$ and $[\![\texttt{"(-)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.n - m$ where $n, m \in \mathbb{Z}$ and thus by $n - m = n + (-m)$ the conclusion holds.
- **Case** $e =$ `"(*)"`: Then $T = a : \mathit{Int} \to b : \mathit{Int} \to \{\nu : \mathit{Int} \mid \nu = a * b\}$ and $[\![\texttt{"(*)"}]\!]_{\Gamma,\Delta'} = \lambda n.\lambda m.n * m$ where $n, m \in \mathbb{Z}$ and thus the conclusion holds.
- **Case** $e =$ `"(//)"`: Then $T = \mathit{Int} \to \{\nu : \mathit{Int} \mid \neg(\nu = 0)\} \to \mathit{Int}$ and

$$[\![\texttt{"(//)"}]\!]_{\Gamma,\Delta'} = s :\Leftrightarrow s = \lambda n.\lambda m. \begin{cases} \left\lfloor \frac{n}{m} \right\rfloor & \text{if } m \neq 0 \\ 0 & \text{else} \end{cases}$$

  where $n, m \in \mathbb{Z}$. Wee see that the "else"-case is dead and the $m \neq 0$-case is well formed. Thus the conclusion holds
- **Case** $e =$ `"if"` e_1 `"then"` e_2 `"else"` e_3 for $e_1, e_2, e_3 \in$ `<exp>`: By the premise of the inference rule we assume $\Gamma, \Delta, \Theta, \vdash e_1 : \mathit{Bool}$, $\Gamma, \Delta, \Theta, \Lambda \cup \{e_1'\} \vdash e_2 : T$ and $\Gamma, \Delta, \Theta, \Lambda \cup \{\neg e_1'\} \vdash e_3 : T$ as well as $e_1 : e_1'$ can be derived for $e_1' \in \mathcal{Q}$. By induction hypothesis $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}(\mathit{Bool})$, $[\![e_2]\!]_{\Gamma,\Delta'} \in \text{value}(T)$ and $[\![e_3]\!]_{\Gamma,\Delta'} \in \text{value}(T)$. Thus, by the definition of the semantics the conclusion holds analogously to the cases above.
- **Case** $e = e_1\ e_2$ for $e_1, e_2 \in$ `<exp>`: By the premise of the inference rule we assume $\Gamma, \Delta, \Theta, \Lambda \vdash e_1 : (a : T_1 \to T_2)$ and $\Gamma, \Delta, \Theta, \Lambda \vdash e_2 : T_1$ as well as $[T_2]_{a \leftarrow e_2'} = T$ and $e_2 : e_2'$ can be derived for $e_2' \in \mathcal{Q}$ and $a \in \mathcal{V}$. Therefore, by the induction hypothesis we know, $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1 \to T_2))$ and $[\![e_1]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T_1))$. Thus $[\![e]\!] \in \text{value}([\hat{T}_2]_{a \leftarrow e_2'})$ and thus the semantics the conclusion holds analogously to the cases above.

- **Case** $e =$ `"\"` a `"->"` e for $a \in \mathcal{V}, e \in$ `<exp>`: Then $T = b : \{\nu : Int \mid r_1\} \to T_2$ for liquid types T_1, T_2 and $b \in \mathcal{V}$. By the premise of the inference rule we assume $\Gamma, \Delta \cup \{(a, \{\nu : Int \mid r_1\})\}, \Theta \cup \{(a, r_1)\}, \Lambda \vdash e : T_2$ can be derived. We now need to show that $[\![e]\!]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\{\nu : Int \mid r_1\} \to T_2)$. We know $[\![e]\!]_{\Gamma,\Delta'} = \lambda b.[\![e]\!]_{\Gamma,\Delta\cup\{(a,b)\}}$ for $b \in \mathcal{V}$. We will therefore by the definition of the abstraction in the lambda expression let $b \in \text{value}_\Gamma(\overline{\Gamma}(\{\nu : Int \mid r_1\}))$ and show $[\![e]\!]_{\Gamma,\Delta'\cup\{(a,b)\}} \in \text{value}_\Gamma(T_2)$. By Theorem 5.4 $\Delta' \cup \{(a, b)\}$ is similar to $\Delta \cup \{(a, \overline{\Gamma}(T_1))\}$ and therefore by induction hypothesis we conclude $[\![e]\!]_{\Gamma,\Delta'\cup\{(a,b)\}} \in \text{value}_\Gamma(T_2)$. Thus the conclusion holds.

- **Case** $e = a$ for $a \in \mathcal{V}$: By the premise of the inference rule we assume $(c, T) \in \Delta$. The semantic requires that there exists an $e \in \text{value}_\Gamma(\overline{\Gamma}(T))$ such that $(c, e) \in \Delta'$. Δ' is similar to Δ and therefore this is a valid assumption. Thus, the semantic is sound and by its definition the conclusion holds analogously to the cases above.

All other cases have been proven in Theorem 5.3. □

4.4. Formulating SMT Statements

So far we have described the inference rules and the subtyping rule. We have yet to give an algorithm that can derive a valid type for a set of given subtyping rules.

Definition 4.1: Liquid Type Variable

We say $\mathcal{K} := \{\kappa_i \mid i \in \mathbb{N}\}$ is the set of all *liquid type variables*.

Note that κ is a special character.

Definition 4.2: Template

We say T is a *template* $:\Leftrightarrow$

$$\begin{aligned} & T \text{ is of form } \{\nu : Int \mid [k]_S\} \\ & \text{where } k \in \mathcal{K} \text{ and } S : \mathcal{V} \nrightarrow \mathcal{Q} \\ \vee\ & T \text{ is of form } a : \{\nu : Int \mid [k]_S\} \to T \\ & \text{where } a \in \mathcal{V}, k \in \mathcal{K}, T \text{ is a template and } S : \mathcal{V} \nrightarrow IntExp. \end{aligned}$$

We define $\mathcal{T}^? := \{T \mid T \text{ is a template}\}$.

A template will be used for a liquid type with unknown refinement. Note that the inference rule for function applications introduces a refinement substitution S. For templates this substitution is not defined and needs to be delayed until the corresponding liquid type has been derived. We will point out whenever the substitution $[k]_S$ will be applied.

Definition 4.3: Type Variable Context

Let $K := \{[k]_S \mid k \in \mathcal{K} \wedge S : \mathcal{V} \nrightarrow \mathit{IntExp}\}$.

—

We say $\Theta : \mathcal{V} \nrightarrow (\mathcal{Q} \cup K)$ is a type variable context.

Our algorithm will resolve a set of suptyping conditions:

Definition 4.4: Subtyping Condition

We say c is a *Subtyping Condition* $:\Leftrightarrow$

c is of form $T_1 <:_{\Theta,\Lambda} T_2$
where T_1, T_2 are liquid types or templates, Θ is a type variable context and $\Lambda \subset \mathcal{Q}$.

We define $\mathcal{C} := \{c \mid c \text{ is a subtyping condition}\}$.

We will also need a function to obtain the set of all liquid type variables of a template or subtyping condition.

Definition 4.5: Vars

We define the following.

$$\begin{aligned}
\text{Vars} : (\mathcal{T} \cup \mathcal{T}^?) &\to \mathcal{P}(\mathcal{K}) \\
\text{Vars}(\{\nu \in \mathit{Int} \mid r\}) &= \{\} \\
\text{Vars}(\{\nu \in \mathit{Int} \mid \kappa_i\}) &= \{\kappa_i\} \\
\text{Vars}(a : \{\nu \in \mathit{Int} \mid \kappa_i\} \to T) &= \{\kappa_i\} \cup \text{Vars}(T)
\end{aligned}$$

$$
\begin{aligned}
\text{Vars} : \mathcal{C} \rightarrow & \mathcal{P}(\mathcal{K}) \\
\text{Vars}(T_1 <:_{\Theta,\Lambda} T_2) = & \text{Vars}(T_1) \cup \text{Vars}(T_2) \\
& \cup \{k \mid (\_, q) \in \Theta \wedge q = [k]_S \text{ for } k \in \mathcal{K} \text{ and } S : \mathcal{V} \nrightarrow \mathit{IntExp}\}
\end{aligned}
$$

The main idea of the algorithm is to first generate a set of predicates and then exclude elements until all subtyping conditions are valid for the remaining predicates. By conjunction over all remaining predicates we result in a valid refinement.

We therefore need a function, depending on a set of variable Q, that will generate a set of predicates. Note that the resulting set should be finite and a subset of $\mathcal{Q}$. If the generated set is too small, then our resulting subtyping conditions might be too weak.

$$
\begin{aligned}
\mathit{Init} : \mathcal{P}(\mathcal{V}) \rightarrow & \mathcal{P}(\mathcal{Q}) \\
\mathit{Init}(V) ::= & \{0 < \nu\} \\
& \cup \{a < \nu \mid a \in V\} \\
& \cup \{\nu < 0\} \\
& \cup \{\nu < a \mid a \in V\} \\
& \cup \{\nu = a \mid a \in V\} \\
& \cup \{\nu = 0\} \\
& \cup \{a < \nu \vee \nu = a \mid a \in V\} \\
& \cup \{\nu < a \vee \nu = a \mid a \in V\} \\
& \cup \{0 < \nu \vee \nu = 0\} \\
& \cup \{\nu < 0 \vee \nu = 0\} \\
& \cup \{\neg(\nu = a) \mid a \in V\} \\
& \cup \{\neg(\nu = 0)\}
\end{aligned}
$$

We can always extend the realm of predicates if the resulting refinements are too weak.

4.4.1. The Inference Algorithm

We will now discuss the inference algorithm. This algorithm inferes a refinements for every template within a set of subtyping conditions.

$$
\begin{aligned}
&\text{Infer} : \mathcal{P}(\mathcal{C}) \to \ (\mathcal{K} \nrightarrow \mathcal{Q})\\
&\text{Infer}(C) =\\
&\quad \text{Let } V := \bigcup_{T_1 <:_{\Theta,\Lambda} T_2 \in C} \{a \mid (a, \_) \in \Theta\}\\
&\qquad Q_0 := \mathit{Init}(V),\\
&\qquad A_0 := \{(\kappa, Q_0) \mid \kappa \in \bigcup_{c \in C} \text{Var}(c)\},\\
&\qquad\ A := \text{Solve}(\bigcup_{c \in C} \text{Split}(c), A_0)\\
&\quad \text{in } \{(\kappa, \bigwedge Q) \mid (\kappa, Q) \in A\}\\
&\quad \text{where } V \subseteq \mathcal{V}, Q_0, Q \subseteq \mathcal{Q}, A_0, A \in \mathcal{K} \nrightarrow \mathcal{Q}, \Theta \text{ is a type variable context and } \Lambda \subseteq \mathcal{Q}.
\end{aligned}
$$

We first split the subtyping conditions for functions into subtyping conditions for simpler templates:

$$
\begin{aligned}
\mathcal{C}^- := \{ \ & \{\nu : \mathit{Int} \mid q_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid q_2\}\\
& \mid (q_1 \in \mathcal{Q} \vee q_1 = [k_1]_{S_1} \text{ for } k_1 \in \mathcal{K}, S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp})\\
& \wedge (q_2 \in \mathcal{Q} \vee q_2 = [k_2]_{S_2} \text{ for } k_2 \in \mathcal{K}, S_2 \in \mathcal{V} \nrightarrow \mathit{IntExp})\}.
\end{aligned}
$$

With this we can now define the Split function.

$$
\begin{aligned}
&\text{Split} : \mathcal{C} \nrightarrow \mathcal{P}(\mathcal{C}^-)\\
&\text{Split}(a : \{\nu : \mathit{Int} \mid q_1\} \to T_2 <:_{\Theta,\Lambda} a : \{\nu : \mathit{Int} \mid q_3\} \to T_4) =\\
&\qquad \{\{\nu : \mathit{Int} \mid q_3\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid q_1\}\} \cup \text{Split}(T_2 <:_{\Theta \cup \{(a,q_3)\},\Lambda} T_4\})\\
&\text{Split}(\{\nu : \mathit{Int} \mid q_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid q_2\}) =\\
&\qquad \{\{\nu : \mathit{Int} \mid q_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid q_2\}\}
\end{aligned}
$$

Note that the result of Split is undefined, if the subtyping condition is not one of the two cases above.

We resolve the obtained subtyping conditions by repeatably checking if a subtyping condition is not valid and removing all predicates that contradict it. By removing the predicate we weaken the resulting refinement.

$\text{Solve} : \mathcal{P}(\mathcal{C}^-) \times (\mathcal{K} \nrightarrow \mathcal{P}(\mathcal{Q})) \to (\mathcal{K} \nrightarrow \mathcal{P}(\mathcal{Q}))$

$\text{Solve}(C, A) =$

Let $S := \{(k, \bigwedge Q) \mid (k, Q) \in A\}$.

If there exists $(\{\nu : \mathit{Int} \mid q_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [k_2]_{S_2}\}) \in C$ such that

$\neg(\forall z \in \mathbb{Z}.\forall i_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_1\}). \ldots \forall i_n \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_n\}).$

$[\![r_1 \wedge p]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![r_2]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})$

for $r_2 := \bigwedge [S(\kappa_2)]_{S_2},\ p := \bigwedge \Lambda,$

$$r_1 := \begin{cases} \bigwedge [S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \text{ and} \\ & S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp} \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases},$$

$$\begin{aligned} \Theta' := \{ & (a, r) \\ & \mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q} \\ & \vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta \\ & \qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\} \end{aligned}$$

$\{(b_1, r'_1), \ldots, (b_n, r'_n)\} = \Theta'$

then $\text{Solve}(C, \text{Weaken}(c, A))$ else A

where $k, k_2 \in \mathcal{K}, S : \mathcal{K} \nrightarrow \mathcal{Q}, Q, \Lambda \subseteq \mathcal{Q}, S_2 : \mathcal{V} \nrightarrow \mathit{IntExp}, q_1 \in \mathcal{K} \cup \mathcal{Q}$,

Θ be a type variable context, $r_1, p, r_2 \in \mathcal{Q}, a \in \mathcal{V}, \Theta' : \mathcal{V} \nrightarrow \mathcal{Q}, r \in \mathcal{Q}, n \in \mathbb{N}, b_i \in \mathcal{V}$,

$r_i \in \mathcal{Q}$ for $i \in \mathbb{N}_0^n$ and $[t]_A$ denotes the substitution for the term t with a substitution A.

Note that we can use an SMT solver to validate

$$\begin{aligned} &\neg(\forall z \in \mathbb{Z}.\forall i_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_1\}). \ldots \forall i_n \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_n\}). \\ &\quad [\![r_1 \wedge p]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![r_2]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}}) \end{aligned}$$

by deciding the satisfiablity of

$$((\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu,b_j)\}}) \wedge r_1 \wedge p) \wedge \neg r_2$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

$$
\begin{aligned}
&\text{Weaken} : \mathcal{C}^- \times (\mathcal{K} \nrightarrow \mathcal{P}(\mathcal{Q})) \nrightarrow (\mathcal{K} \nrightarrow \mathcal{P}(\mathcal{Q}))\\
&\text{Weaken}(\{\nu : \mathit{Int} \mid x\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [k_2]_{S_2}\}, A) =\\
&\quad \text{Let } S := \{(k, \bigwedge Q) \mid (k, Q) \in A\},\\
&\qquad r_1 := \begin{cases} \bigwedge [S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \text{ and } S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp} \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases},\\
&\qquad p := \bigwedge \{[q]_S \mid q \in \Lambda\},\\
&\qquad \Theta' := \{\ (a, r)\\
&\qquad\qquad \mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q}\\
&\qquad\qquad \vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta\\
&\qquad\qquad\qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\}\\
&\qquad \{(b_1, r'_1), \ldots, (b_n, r'_n)\} = \Theta'\\
&\qquad Q_2 := \{\ q\\
&\qquad\qquad \mid q \in A(k_2) \wedge \text{wellFormed}(q, \{(b_1, \{\nu : \mathit{Int} \mid r'_1\}), \ldots, (b_n, \{\nu : \mathit{Int} \mid r'_n\})\})\\
&\qquad\qquad \wedge(\forall z \in \mathbb{Z}.\forall i_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_1\}). \ldots \forall i_n \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_n\}).\\
&\qquad\qquad\qquad [\![r_1 \wedge p]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![[q]_{S_2}]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})\}\\
&\quad \text{in } \{(k, Q) \mid (k, Q) \in A \wedge k \neq k_2\} \cup \{(k_2, Q_2)\}\\
&\quad \text{where } k, k_1 \in \mathcal{K}, Q, Q_2 \subseteq \mathcal{Q}, S : \mathcal{K} \nrightarrow \mathcal{Q}, r_1 \in \mathcal{Q}, p \in \mathcal{Q}, S_2 : \mathcal{V} \nrightarrow \mathit{IntExp}, \Theta' : \mathcal{V} \nrightarrow \mathcal{T},\\
&\quad a \in \mathcal{V}, T' \in \mathcal{T} \cup \mathcal{T}^? n \in \mathbb{N}, b_i \in \mathcal{V}, T_i \in \mathcal{T} \text{ for } i \in \mathbb{N}_0^n \text{ and } [t]_A \text{ denotes the}\\
&\quad \text{substitution for the term } t \text{ with a substitution } A.
\end{aligned}
$$

Note that we can use an SMT solver to validate

$$
\begin{aligned}
&\forall z \in \mathbb{Z}.\forall i_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_1\}). \ldots \forall i_n \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_n\}).\\
&\quad [\![r_1 \wedge p]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![[q]_{S_2}]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}}
\end{aligned}
$$

To do so, we first need to compute $r_2 := [q]_{S_2}$, with that we can now use an SMT solver to decide the satisfiablity of

$$
\neg((\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu,b_j)\}}) \wedge r_1 \wedge p) \vee r_2
$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

Example 4.1

Assume that we have given the following suptyping conditions:

$$
\begin{aligned}
\Theta &:= \{(a, \{Int \mid \kappa_1\}), (b, \{Int \mid \kappa_2\})\} \\
C_0 &:= \{\{\nu : Int \mid \nu = b\} <:_{\Theta,\{a<b\}} \{\nu : Int \mid \kappa_3\}, \\
&\qquad \{\nu : Int \mid \nu = a\} <:_{\Theta,\{\neg(a<b)\}} \{\nu : Int \mid \kappa_3\}, \\
&\qquad a : \{\nu : Int \mid \kappa_1\} \to b : \{\nu : Int \mid \kappa_2\} \to \{\nu : Int \mid \kappa_3\} \\
&\qquad\quad <:_{\{\},\{\}} a : \{\nu : Int \mid True\} \to b : \{\nu : Int \mid True\} \to \{\nu : Int \mid \kappa_4\}
\end{aligned}
$$

Then $V := \{a, b\}$ and $A_0 := \{(\kappa_1, Init(V)), (\kappa_2, Init(V)), (\kappa_3, Init(V)), (\kappa_4, Init(V))\}$.

Splitting the Conditions

We will only consider the last subtyping condition of C_0, all other conditions do not need to be split.

$$
\begin{aligned}
&\text{Split}(a : \{\nu : Int \mid \kappa_1\} \to b : \{\nu : Int \mid \kappa_2\} \to \{\nu : Int \mid \kappa_3\} \\
&\quad <:_{\{\},\{\}} a : \{\nu : Int \mid True\} \to b : \{\nu : Int \mid True\} \to \{\nu : Int \mid \kappa_4\}) \\
&= \text{Split}(a : \{\nu : Int \mid \kappa_1\} <:_{\{\},\{\}} a : \{\nu : Int \mid True\}) \\
&\quad \cup \text{Split}(b : \{\nu : Int \mid \kappa_2\} \to \{\nu : Int \mid \kappa_3\} \\
&\qquad <:_{\{(a,\{\nu:Int \mid True\})\},\{\}} b : \{\nu : Int \mid True\} \to \{\nu : Int \mid \kappa_4\}) \\
&= \{a : \{\nu : Int \mid True\} <:_{\{\},\{\}} a : \{\nu : Int \mid \kappa_1\}\} \\
&\quad \cup \text{Split}(b : \{\nu : Int \mid True\} <:_{\{(a,\{\nu:Int \mid True\})\},\{\}} b : \{\nu : Int \mid \kappa_2\}) \\
&\quad \cup \text{Split}(\{\nu : Int \mid \kappa_3\} <:_{\Theta,\{\}} \{\nu : Int \mid \kappa_4\}) \\
&= \{\{\nu : Int \mid True\} <:_{\{\},\{\}} \{\nu : Int \mid \kappa_1\}, \\
&\quad \{\nu : Int \mid True\} <:_{\{(a,\{\nu:Int \mid True\})\},\{\}} \{\nu : Int \mid \kappa_2\}, \\
&\quad \{\nu : Int \mid \kappa_3\} <:_{\{\Theta\},\{\}} \{\nu : Int \mid \kappa_4\}\}
\end{aligned}
$$

So in conclusion we have the following set of subtypings conditions:

$$
\begin{aligned}
C := \{&\{\nu : Int \mid \nu = b\} <:_{\Theta,\{a<b\}} \{\nu : Int \mid \kappa_3\}, \\
&\{\nu : Int \mid \nu = a\} <:_{\Theta,\{\neg(a<b)\}} \{\nu : Int \mid \kappa_3\}, \\
&\{\nu : Int \mid True\} <:_{\{\},\{\}} \{\nu : Int \mid \kappa_1\}, \\
&\{\nu : Int \mid True\} <:_{\{(a,\{\nu:Int \mid True\})\},\{\}} \{\nu : Int \mid \kappa_2\}, \\
&\{\nu : Int \mid \kappa_3\} <:_{\Theta,\{\}} \{\nu : Int \mid \kappa_4\}\}
\end{aligned}
$$

We therefore now will go through each condition $c \in C$ and check its validity.

Iteration 1, Case $c = \{\nu : Int \mid \nu = b\} <:_{\Theta,\{a<b\}} \{\nu : Int \mid \kappa_3\}$:

We define $S := \{(\kappa_1, \bigwedge Init(V)), (\kappa_2, \bigwedge Init(V)), (\kappa_3, \bigwedge Init(V)), (\kappa_4, \bigwedge Init(V))\}$.

$Init(V)$ contains $\nu = 0$ and $\neg\nu = 0$, so we know that $\bigwedge Init(V)$ can be simplified to *False*.

We now check if

$$\begin{aligned}
&\forall a \in \text{values}_{\{\}}(\{\nu : Int \mid False\}).\\
&\forall b \in \text{values}_{\{\}}(\{\nu : Int \mid False\}).\\
&\forall \nu \in \text{values}_{\{\}}(\{\nu : Int \mid True\}).\\
&\nu = b \wedge a < b\\
&\vDash \forall a \in \text{values}_{\{\}}(\{\nu : Int \mid False\}).\\
&\quad \forall b \in \text{values}_{\{\}}(\{\nu : Int \mid False\}).\\
&\quad \forall \nu \in \text{values}_{\{\}}(\{\nu : Int \mid True\}).\\
&\quad False
\end{aligned}$$

is not valid.

We know that $\text{values}_{\{\}}(\{\nu : False\}) = \{\}$, and therefore this can be simplified to $True \vDash True$, which is valid.

Iteration 1, Case $c = \{\nu : Int \mid \nu = a\} <:_{\Theta,\{\neg(a<b)\}} \{\nu : Int \mid \kappa_3\}$:

The argument is analogously to the previous case.

Iteration 1, Case $c = \{\nu : Int \mid True\} <:_{\{\},\{\}} \{\nu : Int \mid \kappa_1\}$:

We now check if

$$\forall \nu \in \text{values}_{\{\}}(\{\nu : Int \mid True\}).True \vDash \forall \nu \in \text{values}_{\{\}}(\{\nu : Int \mid True\}).False$$

is valid. This time we can ignore the quantifiers and thus it simplifies to $True \vDash False$, which is not valid.

We therefore will now weaken A_0. To do so we compute all $q \in A_0(\kappa_1)$ such that wellFormed(q) and

$$\forall \nu \in \text{values}_{\{\}}(\{\nu : Int \mid True\}).[\![True]\!]_{\{\}} \vDash \forall \nu \in \text{values}_{\{\}}(\{\nu : Int \mid True\}).[\![q]\!]_{\{\}}.$$

There are only two values for q that are well formed: *True* and *False*.

The resulting set is $Q_2 := \{True\}$ and thus we replace A_0 with

$$A := \{(\kappa_1, \{\mathit{True}\}), (\kappa_2, \mathit{Init}(V)), (\kappa_3, \mathit{Init}(V)), (\kappa_4, \mathit{Init}(V))\}$$

Iteration 1, Case $c = \{\nu : \mathit{Int} \mid \mathit{True}\} <:_{\{(a,\{\nu:\mathit{Int} \mid \mathit{True}\})\},\{\}} \{\nu : \mathit{Int} \mid \kappa_2\}$:

The argument is analogously to the previous case, resulting in the updated value for A:

$$A = \{(\kappa_1, \{\mathit{True}\}), (\kappa_2, \{\mathit{True}\}), (\kappa_3, \mathit{Init}(V)), (\kappa_4, \mathit{Init}(V))\}$$

Iteration 1, Case $c = \{\nu : \mathit{Int} \mid \kappa_3\} <:_{\Theta,\{\}} \{\nu : \mathit{Int} \mid \kappa_4\}\}$:

The suptyping condition is valid, analogously to the first case of this iteration.

Iteration 2, Case $c = \{\nu : \mathit{Int} \mid \nu = b\} <:_{\Theta,\{a<b\}} \{\nu : \mathit{Int} \mid \kappa_3\}$:

We check the validity of

$$\begin{aligned}
&\forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\nu = b \wedge a < b\\
&\vDash \forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \mathit{False}.
\end{aligned}$$

It is easy to see, that it is not valid.

Thus we now compute all $q \in A(\kappa_3)$ such that wellFormed(q) and

$$\begin{aligned}
&\forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\nu = b \wedge a < b\\
&\vDash \forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad q.
\end{aligned}$$

is valid. The resulting set Q_2 is the following.

$$Q_2 := \{a < \nu, \nu = b, \neg(\nu = a), \nu < b \vee \nu = b, b < \nu \vee \nu = b, \nu < a \vee \nu = a, \\ a < \nu \vee \nu = a\}$$

Therefore we update A:

$$\begin{aligned} A = \{&(\kappa_1, \{\mathit{True}\}), (\kappa_2, \{\mathit{True}\}), \\ &(\kappa_3, \{a < \nu, \nu = b, \neg(\nu = a), \nu < b \vee \nu = b, b < \nu \vee \nu = b, \nu < a \vee \nu = a, \\ &\quad a < \nu \vee \nu = a\}), \\ &(\kappa_4, \mathit{Init}(V))\} \end{aligned}$$

Iteration 2, Case $c = \{\nu : \mathit{Int} \mid \nu = a\} <:_{\Theta, \{\neg(a<b)\}} \{\nu : \mathit{Int} \mid \kappa_3\}$:

We check the validity of

$$\begin{aligned} &\forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\nu = a \wedge \neg(a < b) \\ &\models \forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\quad \forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\quad \forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\quad a < \nu \wedge \nu = b \wedge \neg(\nu = a) \wedge (\nu < b \vee \nu = b) \\ &\qquad \wedge (b < \nu \vee \nu = b) \wedge (\nu < a \vee \nu = a) \wedge (a < \nu \vee \nu = a). \end{aligned}$$

It is not valid, because $\nu = a \wedge \neg(a < b) \models \nu = b$ is not valid.

Thus we compute all $q \in A(\kappa_3)$ such that wellFromed(q) and

$$\begin{aligned} &\forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\nu = a \wedge \neg(a < b) \\ &\models \forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\quad \forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\quad \forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}). \\ &\quad q. \end{aligned}$$

is valid. The resulting set Q_2 is the following.

$$Q_2 := \{b < \nu \vee \nu = b, a < \nu \vee \nu = a\}$$

Thus we update A:

$$A = \{(\kappa_1, \{\mathit{True}\}), (\kappa_2, \{\mathit{True}\}), \quad (\kappa_3, \{b < \nu \vee \nu = b, a < \nu \vee \nu = a\}), (\kappa_4, \mathit{Init}(V))\}$$

Iteration 2, Case $\{\nu : \mathit{Int} \mid \mathit{True}\} <:_{\{\},\{\}} \{\nu : \mathit{Int} \mid \kappa_1\}$:

Nothing has changed since the last iteration, therefore this case can be skipped.

Iteration 2, Case $\{\nu : \mathit{Int} \mid \mathit{True}\} <:_{\{(a,\{\nu:\mathit{Int} \mid \mathit{True}\})\},\{\}} \{\nu : \mathit{Int} \mid \kappa_2\}$:

The argument is analogously to the previous case, therefore this case can be skipped.

Iteration 2, Case : $\{\nu : \mathit{Int} \mid \kappa_3\} <:_{\Theta,\{\}} \{\nu : \mathit{Int} \mid \kappa_4\}$:

We check the validity of

$$\begin{aligned}
&\forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\{b < \nu \vee \nu = b \wedge a < \nu \vee \nu = a\}\\
&\models \forall a \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \forall b \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \forall \nu \in \text{values}_{\{\}}(\{\nu : \mathit{Int} \mid \mathit{True}\}).\\
&\quad \mathit{False}.
\end{aligned}$$

We see that this is not valid, therefore we derive the new set Q_2. Note that $A(\kappa_3) \subseteq \mathit{Init}(V)$ and therefore $Q_2 = A(\kappa_3)$.

We update the corresponding entry in A:

$$\begin{aligned}
A = \{&(\kappa_1, \{\mathit{True}\}), (\kappa_2, \{\mathit{True}\}),\\
&(\kappa_3, \{b < \nu \vee \nu = b, a < \nu \vee \nu = a\}),\\
&(\kappa_4, \{b < \nu \vee \nu = b, a < \nu \vee \nu = a\})\}
\end{aligned}$$

Iteration 3:

In this iteration all subtyping conditions are valid, thus the algorithm stops. The resulting set of substitutions is therefore the following

$$\begin{aligned}&\{(\kappa_1, \mathit{True}), (\kappa_2, \mathit{True}),\\ &\ (\kappa_3, (b < \nu \vee \nu = b) \wedge (a < \nu \vee \nu = a)),\\ &\ (\kappa_4, (b < \nu \vee \nu = b) \wedge (a < \nu \vee \nu = a))\}\end{aligned}$$

4.4.2. Correctness

We now show the correctnes of the algorithm.

The algorithm that we described can fail if the subtyping conditions are not well-formed.

Definition 4.6: Well-formed Subtyping Condition

We say a subtyping condition c is well-formed if the following holds

$$\begin{aligned}&c \text{ has the form } \{\nu : \mathit{Int} \mid [k_1]_{S_1}\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [k_2]_{S_2}\}\\ \vee\ &c \text{ has the form } \{\nu : \mathit{Int} \mid r\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [k_2]_{S_2}\}\\ \vee\ &c \text{ has the form } \{\nu : \mathit{Int} \mid [k_1]_{S_1}\} \to T_1 <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid r\} \to T_2\\ &\quad \text{such that } T_1 <:_{\Theta,\Lambda} T_2 \text{ is well-formed.}\end{aligned}$$

where $r \in \mathcal{Q}$,$k_1, k_2 \in \mathcal{K}$, $S_1, S_2 : \mathcal{V} \nrightarrow \mathit{IntExp}$,$\Theta$ is a type variable context, $\Lambda \subset \mathcal{Q}$ and $T_1, T_2 \in (\mathcal{T} \cup \mathcal{T}^?)$

We will prove that for a given set of well-formed subtyping conditions the algorithm generates refinements that satisfy the conditions. Addtionally we will show that the conditions are the sharpest possible conditions that can be generated from predicates in *Init*.

Theorem 4.1

Let C be a set of well-formed conditions, $S := \mathrm{Infer}(C)$ and $V := \bigcup_{\hat{T}_1 <:_{\Theta,\Lambda} \hat{T}_2 \in C} \{a \mid (a, \_) \in \Theta\}$.

—

For every subtyping condition $(T_1 <:_{\Theta,\Lambda} T_2) \in C$, let

$$\begin{aligned}\Theta' := \{\ & (a, r) \\ & \mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q} \\ & \vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta \\ & \qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\}\end{aligned}$$

and $\{(b_1, r'_1), \ldots, (b_n, r'_n)\} := \Theta'$. Then we have the following correctness property:

$$\begin{aligned}&[T_1]_S \in \mathcal{T} \wedge [T_2]_S \in \mathcal{T} \\ &\wedge\ [T_1]_S <:_{\Theta',\Lambda} [T_2]_S \\ &\wedge\ \forall S' \in (\mathcal{V} \to \mathcal{Q}).(\forall a \in \mathcal{V}.S'(a) \text{ is well defined} \Rightarrow \exists Q \in \mathit{Init}(V).S'(a) = \bigwedge Q) \\ &\quad \wedge [T_1]_{S'} \in \mathcal{T} \wedge [T_2]_{S'} \in \mathcal{T} \\ &\quad \wedge ([T_1]_{S'} <:_{\Theta',\Lambda} [T_2]_{S'} \\ &\qquad \Rightarrow (\forall a \in \mathcal{V}.S(a) \text{ and } S'(a) \text{ are well defined} \\ &\qquad\quad \Rightarrow (\forall \nu \in \mathbb{Z}.\forall i_1 \in \mathrm{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_1\}).\ldots.\forall i_n \in \mathrm{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_n\}). \\ &\qquad\qquad [\![S(a)]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![S'(a)]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}}))))\end{aligned}$$

The first post-condition states that $[T_1]_S$ and $[T_2]_S$ are not templates. The second condition states that S is a solution, meaning that $[T_1]_S$ is a subtype of $[T_2]_S$. The third condition states that S is the sharpest solution.

Proof. Let C be a set of well-formed conditions, $S := \mathrm{Infer}(C)$ and $V := \bigcup_{T_1 <:_{\Theta,\Lambda} T_2 \in C} \{a \mid (a, \_) \in \Theta\}$. Let $Q_0 := \mathit{Init}(V)$, $A_0 := \{(\kappa, Q_0) \mid \kappa \in \bigcup_{c \in C} \mathrm{Var}(c)\}$ and $C_1 := \bigcup_{c \in C} \mathrm{Split}(c)$. Then by Theorem 4.2 (given below) $C_1 \subset \mathcal{C}^-$ is a set of well-formed conditions. Let $A := \mathrm{Solve}(C_1, A_0)$ and therefore $S = \{(\kappa, \bigwedge Q) \mid (\kappa, Q) \in A\}$. Then by Theorem 4.3 (given below) the conclusion holds. □

Theorem 4.2

Let c is a well-formed condition and $C := \mathrm{Split}(c)$.

—

Then $C \subseteq \mathcal{C}^-$ and, for every $c \in C$, c is a well-formed condition.

Proof. Let c is a well-formed condition. Then $C := \mathrm{Split}(c)$ is well-defined. We prove the theorem by induction on well-formed conditions.

Case c $= \{\nu : Int \mid q\} <:_{\Theta,\Lambda} \{\nu : Int \mid [k_2]_{S_2}\}$: c is well-formed and $c \in C^-$. Then $C = c$ and therefore the conclusion holds.

Case c $= \{\nu : Int \mid [k_1]_{S_1}\} \to T_1 <:_{\Theta,\Lambda} \{\nu : Int \mid r\} \to T_2$ such that $T_1 <:_{\Theta,\Lambda} T_2$ is well-formed: Let $C_0 := \text{Split}(T_1 <:_{\Theta,\Lambda} T_2)$. By the induction hypothesis $C_0 \subseteq \mathcal{C}^-$ and for every $c \in C_0$, c is a well-formed condition. Therefore, by appling Theorem 4.1 to each element in $C := \{\{\nu : Int \mid r\} <:_{\Theta,\Lambda} \{\nu : Int \mid [k_1]_{S_1}\}\} \cup C_0$, the conlusion holds. □

Theorem 4.3

Let $C \subseteq \mathcal{C}^-$ be a set of well-formed conditions, $A_1, A_2 : \mathcal{K} \nrightarrow \mathcal{Q}$ and $V := \bigcup_{T_1 <:_{\Theta,\Lambda} T_2 \in C} \{a \mid (a, \_) \in \Theta\}$. Let for all $a \in V$, $A_1(a)$ be well defined. Let $A_2 = \text{Solve}(C, A_1)$ and $S = \{(\kappa, \bigwedge Q) \mid (\kappa, Q) \in A_2\}$.

—

Then for every $a \in V$, $A_2(a) \subseteq A_1(a)$.

For every subtyping condition $(T_1 <:_{\Theta,\Lambda} T_2) \in C$, let

$$\begin{aligned}
\Theta' := \{\ & (a, r) \\
& \mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q} \\
& \vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta \\
& \qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\}
\end{aligned}$$

and $\{(b_1, r'_1), \ldots, (b_n, r'_n)\} = \Theta'$. We then have the following correctness property.

$$\begin{aligned}
& [T_1]_S \in \mathcal{T} \wedge [T_2]_S \in \mathcal{T} \\
\wedge\ & [T_1]_S <:_{\Theta',\Lambda} [T_2]_S \\
\wedge\ & \forall S' \in (\mathcal{V} \to \mathcal{Q}).(\forall a \in V.\exists Q \in \mathcal{P}(A_1(a)).S'(a) = \bigwedge Q) \\
& \quad \wedge [T_1]_{S'} \in \mathcal{T} \wedge [T_2]_{S'} \in \mathcal{T} \\
& \quad \wedge ([T_1]_{S'} <:_{\Theta',\Lambda} [T_2]_{S'} \\
& \qquad \Rightarrow \forall a \in V.\forall \nu \in \mathbb{Z}.\forall i_1 \in \text{value}_\Gamma(\{\nu : Int \mid r'_1\}). \ldots \forall i_n \in \text{value}_\Gamma(\{\nu : Int \mid r'_n\}). \\
& \qquad\quad [\![S(a)]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![S'(a)]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})
\end{aligned}$$

The post-condition is the same as for Infer.

Proof. Let $C \subseteq \mathcal{C}^-$ be a set of well-formed conditions, $A_1, A_2 : \mathcal{K} \nrightarrow \mathcal{Q}$ and $V := \bigcup_{T_1 <:_{\Theta,\Lambda} T_2 \in C} \{a \mid (a, \_) \in \Theta\}$. Let for all $a \in V$, $A_1(a)$ be well defined. Let $A_2 = \text{Solve}(C, A_1)$ and $S = \{(\kappa, \bigwedge Q) \mid (\kappa, Q) \in A_2\}$.

Show that the then-branch ensures the postcondition

So let $c \in C$ be a condition, such that the if-condition holds. We know that $A_2 = \text{Solve}(C, \text{Weaken}(c, A_1))$. We know that c is a wellformed condition and for all $a \in V$, $A_1(a)$ is defined and therfore the precondition of $\text{Weaken}(c, A_1)$ holds. By Theorem 4.4 $A_2(a)$ is defined for all a for which A_1 is defined, namely for all $a \in V$, thus the precondition of $\text{Solve}(C, \text{Weaken}(c, A_1))$ holds and therefore the postcondition of $\text{Solve}(C, \text{Weaken}(c, A_1))$ holds. In particular we know that for every $a \in V$, $A_2(a) \subseteq \text{Weaken}(c, A_1)$. By Theorem 4.4 we also know that for every $a \in V$, $\text{Weaken}(c, A_1)(a) \subseteq A_1(a)$. Thus for every $a \in V$,$A_2(a) \subseteq A_1(a)$.

We have left to show that for every $a \in V$, if the sharpest solution is generated by subsets of $A_2(a)$, than it is also the sharpest solution over all subsets of $A_1(a)$.

By Theorem 4.4 (given below) we know that for every $a \in V$, $A_2(a)$ contains the smallest subset of $A_1(a)$ such that the properies hold. Thus $A_2(a)$ is its definition the sharpest solution over all subsets of $A_1(a)$.

Show that the else-branch ensures the postcondition

Once the recursion is done, we can assume that for all $c \in C$, c has the form $(\{\nu : \mathit{Int} \mid q_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [k_2]_{S_2}\})$ and the following holds:

$$\begin{aligned}&(\forall z \in \mathbb{Z}.\forall i_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r_1'\}).\ldots\forall i_n \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r_n'\}).\\ &\quad [\![r_1 \wedge p]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![r_2]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})\end{aligned}$$

for

$$\begin{aligned}
&r_2 := \bigwedge [S(\kappa_2)]_{S_2},\ p := \bigwedge \Lambda,\\
&r_1 := \begin{cases} \bigwedge [S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \text{ and } S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp} \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases},\\
&\Theta' := \{\ (a, r)\\
&\qquad \mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q}\\
&\qquad \vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta\\
&\qquad\qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\}\\
&\{(b_1, r_1'), \ldots, (b_n, r_n')\} = \Theta'.
\end{aligned}$$

where $k, k_2 \in \mathcal{K}, S : \mathcal{K} \nrightarrow \mathcal{P}(\mathcal{Q}), Q, \Lambda \subseteq \mathcal{Q}, S_2 : \mathcal{V} \nrightarrow \mathit{IntExp}, q_1 \in \mathcal{K} \cup \mathcal{Q}, \Theta$ be a type variable context $, r_1, p, r_2 \in \mathcal{Q}, a \in \mathcal{V}, \Theta' : \mathcal{V} \nrightarrow \mathcal{Q}, r \in \mathcal{Q}, n \in \mathbb{N}, b_i \in \mathcal{V}, r_i \in \mathcal{Q}$ for $i \in \mathbb{N}_0^n$ and $[t]_A$ denotes the substitution for the term t with a substitution A. This is the same as saying $[T_1]_S <:_{\Theta',\Lambda} [T_2]_S$ if $[T_1]_S \in \mathcal{T}$ and $[T_2]_S \in \mathcal{T}$. We know by Theorem 4.4 that $A_2(a)$ is defined for all $a \in V$ and thus $[T_1]_S \in \mathcal{T} \wedge [T_2]_S \in \mathcal{T}$.

We still need to show that it is the sharpest solution. To do so, we will sketch a proof by contradiction. Assume there exists S' and $a \in V$ such that $S'(a)$ is sharper then $S(a)$. If $S(a) = \mathit{False}$ then there can not exist a sharper refinement $S'(a)$ and therefore this is a contradiction. If $S(a) \neq \mathit{False}$ then it must have been produced by calling Weaken. This is a contradiction, as by Theorem 4.4 we know that Weaken produces the sharpest solution therefore $S'(a) = S(a)$. $\square$

Theorem 4.4

Let $c \in \mathcal{C}^-$ be a well-formed condition and therefore c has the form $\{\nu : \mathit{Int} \mid x\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [k_2]_{S_2}\}$ where x has the form $[k_1]_{S_1}$ or r. Let $A_1, A_2 : \mathcal{K} \nrightarrow \mathcal{Q}$. Let for all $a \in V$, $A_1(a)$ be well defined and $A_2 = \text{Weaken}(c, A_1)$. Let $S := \{(k, \bigwedge Q) \mid (k, Q) \in A_2\}$ and

$$r_1 := \begin{cases} \bigwedge[S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \\ & \text{for } k \in \mathcal{K} \\ & \text{and } S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp} \\ q_1 & \text{if } q_1 \in \mathcal{Q}. \end{cases}$$

—

Then the following holds.

$$\begin{aligned}
&(\forall k \neq k_2 . A_1(k) = A_2(k)) \\
\wedge\ & A_2(k_2) \subseteq A_1(k_2) \\
\wedge\ & [k_2]_S \in \mathcal{Q} \wedge \{\nu : \mathit{Int} \mid r_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [[k_2]_S]_{S_2}\} \\
\wedge\ & \forall A_2' : \mathcal{V} \nrightarrow \mathcal{P}(\mathcal{Q}).(\forall k \neq k_2 . A_1(k) = A_2'(k)) \\
& \quad \wedge\ A_2'(k_2) \subseteq A_1(k_2) \\
& \quad \wedge \text{Let } S' := \{(k, \bigwedge Q) \mid (k, Q) \in A_2'\}, \\
& \qquad r_1' := \begin{cases} \bigwedge[S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases} \\
& \quad \text{in } [k_2]_{S'} \in \mathcal{Q} \\
& \qquad \wedge \{\nu : \mathit{Int} \mid r_1'\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [[k_2]_{S'}]_{S_2}\} \Rightarrow A_2' \subseteq A_2
\end{aligned}$$

The first post-condition states that by updating A_1 to A_2 only the value for k_2 changes. The second condition states that the updated value $A_2(k_2)$ needs to be a subset of the old value $A_1(k_2)$. The third condition states that the resulting substitution S generates a refinement $[k_1]_S$ that makes the subtyping condition valid.

The fourth condition states that the value $A_2(k_2)$ is the smallest subset of $A_1(k_2)$ such that the previous conditions hold.

Proof. Let $c \in \mathcal{C}^-$ be a well-formed condition and therefore c has the form $\{\nu : \mathit{Int} \mid x\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [k_2]_{S_2}\}$ where x has the form $[k_1]_{S_1}$ or r. Let $A_1, A_2 : \mathcal{K} \nrightarrow \mathcal{Q}$. Let for all $a \in V$, $A_1(a)$ be well defined and $A_2 = \text{Weaken}(c, A_1)$. Let

$$
\begin{aligned}
&S := \{(k, \bigwedge Q) \mid (k, Q) \in A_2\} \\
&r_1 := \begin{cases} \bigwedge [S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \text{ and } S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp} \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases} \\
&p := \bigwedge \{[q]_S \mid q \in \Lambda\}, \\
&\Theta' := \{\ (a, r) \\
&\qquad \mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q} \\
&\qquad \vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta \\
&\qquad\qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\} \\
&\{(b_1, r'_1), \ldots, (b_n, r'_n)\} = \Theta' \\
&Q_2 := \{\ q \\
&\qquad \mid q \in A(k_2) \wedge \text{wellFormed}(q, \{(b_1, \{\nu : \mathit{Int} \mid r'_1\}), \ldots, (b_n, \{\nu : \mathit{Int} \mid r'_n\})\}) \\
&\qquad \wedge (\forall z \in \mathbb{Z}. \forall i_1 \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_1\}). \ldots \forall i_n \in \text{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_n\}). \\
&\qquad\qquad [\![r_1 \wedge p]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![[q]_{S_2}]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})\}
\end{aligned}
$$

where $k, k_1 \in \mathcal{K}, Q, Q_2 \subseteq \mathcal{Q}, S : \mathcal{K} \nrightarrow \mathcal{P}(\mathcal{Q}), r_1 \in \mathcal{Q}, p \in \mathcal{Q}, S_2 : \mathcal{V} \nrightarrow \mathit{IntExp}, \Theta' : \mathcal{V} \nrightarrow \mathcal{T}, a \in \mathcal{V}, T' \in \mathcal{T} \cup \mathcal{T}^? n \in \mathbb{N}, b_i \in \mathcal{V}, T_i \in \mathcal{T}$ for $i \in \mathbb{N}_0^n$ and $[t]_A$ denotes the substitution for the term t with a substitution A.

Then

$$A_2 = \{(k, Q) \mid (k, Q) \in A \wedge k \neq k_2\} \cup \{(k_2, Q_2)\}$$

and therefore $\forall k \neq k_2. A_1(k) = A_2(k)$.

We know $A_2(k_2) = Q_2$ and $Q_2 \subseteq A_1(k_2)$ and therefore $A_2(k_2) \subseteq A_1(k_2)$.

We know by the definition of S that $S(k_2) = \bigwedge Q_2$. Therefore it is easy to see that $\{\nu : \mathit{Int} \mid r_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [[k_2]_S]_{S_2}\}$ is true because by definiton of Q_2 it is true for all $q \in Q_2$, thus it is true for $\bigwedge Q_2$ as well.

We finish off by proving that the result is the sharpest, meaning that A_2 is the smallest subset such hat the premise holds. So let there exists a A'_2 such that $\forall k \neq k_2. A_1(k) = A'_2(k)$, $A'_2(k_2) \subseteq A_1(k_2)$,$[k_2]_{S'} \in \mathcal{Q}$,$\{\nu : \mathit{Int} \mid r'_1\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid [[k_2]_{S'}]_{S_2}\}$ and $A'_2 \subseteq A_2$ for given S' and r_1.

This means for every $q \in A_2'(k_2)$ that $q \in A_1(k_2)$ and by the definition of subtyping, $\text{wellFormed}(q, \{(b_1, \{\nu : Int \mid r_1'\}), \ldots, (b_n, \{\nu : Int \mid r_n'\})\})$ and

$$\forall z \in \mathbb{Z}. \forall i_1 \in \text{value}_\Gamma(\{\nu : Int \mid r_1'\}). \ldots \forall i_n \in \text{value}_\Gamma(\{\nu : Int \mid r_n'\}).$$
$$[\![r_1 \wedge p]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [\![[q]_{S_2}]\!]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}}.$$

By the definition of A_2, this means that $q \in A_2(k_2)$. Thus A_2 is the smallest subset. $\square$

5. Implementation

We will now discuss the implementation of the Elm type system and the implementation of the refinement types. We use these implementations for rapid prototyping.

Section 5.1 will discuss the implementation of the Elm type system in the software system "K Framework". In Section 5.2 we will go over the implementation of the refinement types in Elm. We could not perform this implementation in the K Framework, as it lacks a way to communicate with an external SMT-Solver. In Section 5.3 we will give a detailed walkthrough of the Elm code. In Section 5.4 we will demonstrate the implemented algorithm on an example code.

5.1. The Elm Type System in the K Framework

The K Framework [RS14] was created in 2003 by Grigore Rosu. It is a research language and a system of tools for designing and formalizing programming languages. These include tools for parsing, execution, type checking and program verification [Ste+16]. Most of the features of the system are performed by rewriting systems that are specified using its programming language called "K Language".

The main usage besides the creation and formalization of new languages is to create formal languages of existing programming languages. These include C [HER15], Java [BR15], JavaScript [PSR15], PHP [FM14], Python [Gut13] and Rust [Kan+18].

The project was pursued by the Formal Systems Laboratory Research Group and the University of Illinois, USA. The software itself is open source while the various more specialized tools are distributed by the company Runtime Verification Inc. These include an analysing tool for C called RV-Match that is based on the formal C language written in K language [Gut+16] and more recently a tool for verifying smart contract written for the crypto-coin Ethereum [Hil+18].

We will be using K Framework (Version 4) to express small step semantics of the denotational semantics from Chapter 4.2.6. We can validate the semantic by letting the K Framework apply the rewriting rules upon some examples.

A file written in the K langauge has a specific structure.

```
require "unification.k"
require "elm-syntax.k"

module ELM-TYPESYSTEM
  imports DOMAINS
  imports ELM-SYNTAX

  configuration <k> $PGM:Exp </k>
                <tenv> .Map </tenv>
  //..

  syntax KResult ::= Type
endmodule
```

One can specify the realm upon which the rewriting system can be executed by using the `configuration` keyword. Here we specify two parts: `<k></k>` containing the expression and `<tenv></tenv>` containing the type context.

We also need to specify the end result using the keyword `KResult`. Once the rewriting system reaches such an expression, it will stop. If not specified the system might not terminate.

5.1.1. Implementing the Formal Language

To implement the formale Elm language in K Framework we need to translate the formal grammar into the K language.

```
syntax Type
  ::= "bool"
    | "int"
    | "{}Type"
    | "{" ListTypeFields "}Type" [strict]
    | Type "->" Type       [strict,right]
    | LowerVar
    | "(" Type ")"              [bracket]
    | ..
```

Additionally, we can include meta-information: `strict` to ensure the inner expression gets evaluated first, `right/left` to state in which direction the expressions should be evaluated and `bracket` for brackets that will be replaced with meta level brackets during paring.

Rules are written as rewriting rules instead of inference rules.

```
syntax Exp ::= Type
rule E1:Type E2:Type
  => E1 =Type (E2 -> ?T:Type)
    ~> ?T
syntax KResult ::= Type
```

The rule itself has the syntax `rule ... => ...` . The inner expressions need to be rewritten into types before the outer rule can be applied. We can include an additional `syntax` line before the rule and a `KResult` to ensure that the rewriting system keeps on applying rules that are written above the `syntax` line until the result is of the form defined in the `syntax` line. Only then it may continue.

Additionally, we have variables starting with an uppercase letter and existentially quantified variables starting with a question mark.

The system itself allows for a more untraditional imperative rewriting system using `~>`. This symbol has only one rule: `rule . ~> A => A` (read as "`. ~> A` can be rewritten into `A`") where "`.`" denotes the empty term. Thus, the left side of a `~>` expression needs to be first rewritten to "`.`" before the right side can be changed. Until then the right side is just a term.

The type system can infer mono types by applying rules as long as possible. For poly types we need to implement the polymorphism, in particular instantiation and the generalization. The inference rules that we have presented in the Section 3.3 are not monomorphic and therefore can not be implemented in this state. We will therefore modify them slightly. In particular, we will implement the Alogrithm J as described in in the original papar by Milner [Mil78].

5.1.2. Implementing Algorithm J

The Algorithm J is an optimized algorithm for implementing polymorphism in a programming language. This algorithm is imperative but is typically presented as logical rules:

$$\frac{a : T_1 \quad T_2 = \mathit{inst}(T_1)}{\Gamma \vdash_J a : T_2} \qquad \text{[Variable]}$$

$$\frac{\Gamma \vdash_J e_0 : T_0 \quad \Gamma \vdash_J e_1 : T_1 \quad T_2 = \mathit{newvar} \quad \mathit{unify}(T_0, T_1 \to T_2)}{\Gamma \vdash_J e_0 e_1 : T_2} \qquad \text{[Call]}$$

$$\frac{T_1 = newvar \quad \Gamma, x : T_1 \vdash_J e : T_2}{\Gamma \vdash_J \backslash x\texttt{->}e : T_0 \to T_1} \qquad \text{[Lambda]}$$

$$\frac{\Delta_1 \vdash_J e_0 : T_1 \quad \Delta_1, a : \text{insert}_{\Delta_1}(\{T_1\}) \vdash_J e_1 : T_2}{\Delta \vdash_J \texttt{let}x\texttt{=}e_0\texttt{in}e_1 : T_2} \qquad \text{[LetIn]}$$

To adjust our own infrence rules, we only need to replace the rules of *let in*, *lambda*, *call* and *variable* with the rules above. The imperative functions are *newvar*, *unify* and *inst*:

- *newvar* creates a new variable.
- *inst* instantiates a type with new variables.
- *uinify* checks whether two types can be unified.

The K Framework has these imperative functions implemented in the `Unification.k` module. In order to use them, we need to first properly define poylmorphic types.

```
syntax PolyType ::= "forall" Set "." Type
```

Next we tell the system that we want to use the unification algorithm on types.

```
syntax Type ::= MetaVariable
```

We can now use the function `#renameMetaKVariables` for *inst* and `?T` for *newvar*.

```
rule <k> variable X:Id => #renameMetaKVariables(T, Tvs) ...</k>
    <tenv>... X |-> forall Tvs . T
    ...</tenv>

rule <k> fun A:Id -> E:Type => ?T:Type -> E ~> setTenv(TEnv) ...</k>
    <tenv> TEnv:Map => TEnv [ A <- ?T ] </tenv>

syntax KItem ::= setTenv(Map)
  rule <k> T:Type ~> (setTenv(TEnv) => .) ...</k>
   <tenv> _ => TEnv </tenv>
```

Note that the `setTenv` function ensures that `?T` is instantiated before it is inserted into the environment.

For implementing the unification we use `#metaKVariables` for getting all bound variables and `#freezeKVariables` to ensure that variables in the environment need to be newly instantiated whenever they get used.

```
rule <k> let X = T:Type in E => E ~> setTenv(TEnv)
    ...</k>
```

```
<tenv> TEnv
  => TEnv[ X
    <- forall
      (#metaKVariables(T) -Set #metaKVariables(setTenv(TEnv))) .
      ( #freezeKVariables(T, setTenv(TEnv)):>Type
      )
  ]
</tenv>
```

As for *unify*, we can take advantage of the build-in pattern matching capabilities:

```
syntax KItem ::= Type "=Type" Type
rule T =Type T => .
```

By using a new function `=Type` with the rewriting rule `rule T =Type T => .` we can force the system to pattern match when ever we need to. Note that if we do not use this trick, the system will think that all existentially quantified variables are type variables and will therefore stop midway.

5.1.3. Example

We will now showcase how the K Framework infers types using the following example:

```
let
  model = []
in
(::) 1 model
```

We first need to write the example into a form that K Framework can parse. Using the following syntax:

```
syntax Exp
    ::= "let" LowerVar "=" Exp "in" Exp                [strict(2)]
      | Exp Exp                                        [left,strict]
      | "[]Exp"
      | "intExp" Int
      | "(::)"
      | "variable" LowerVar
      | ..
```

Translating the program into our K Framework syntax, this results in the following file:

```
<k>
```

```
let
  model = []Exp
in
((::) (intExp 1)) (variable model)
</k>
<tenv> .Map </tenv>
```

Here `.Map` denotes the empty type context. Also note that we have already applied the `left` rule by adding brackets. The K Framework uses this rule at parse time, so this is just syntax sugar.

The K Framework will now walk through the abstract syntax tree to find the first term it can match. By specifying `strict(2)` we tell the system that `let in` can only be matched once `[]Exp` is rewritten.

By appling the rule

```
rule []Exp => list ?A:Type
```

K-Framework obtains the following result.

```
<k>
let
  model = list ?A0:Type
in
((::) (intExp 1)) (variable model)
</k>
<tenv> .Map </tenv>
```

The system remembers the type hole `?A0` and will fill it in as soon as it finds a candidate for it. By using the rule

```
rule <k> let X = T:Type in E => E ~> setTenv(TEnv)
    ...</k>
    <tenv> TEnv
      => TEnv[ X
        <- forall
          (#metaKVariables(T) -Set #metaKVariables(setTenv(TEnv)))
          .
          ( #freezeKVariables(T, setTenv(TEnv)):>Type
          )
      ]
    </tenv>
```

the system rewrites the `let in` expression.

Thus our example gets rewritten into the following.

```
<k>
((::) (intExp 1)) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Note that we have just witnessed the generalization being applied: The free variable `?A` of the type got bound resulting in the poly type `forall A0 . (list (#freeze(A0))`. These poly types only exist inside the type environment.

The rule for function application is strict, we therefore need to first rewrite `(::) (intExp 1)` and `variable model`. By appling the rules

```
rule (::) => ?A:Type -> ( list ?A ) -> ( list ?A )

rule intExp I:Int => int
```

the left expression can be rewritten.

```
<k>
((?A1:Type -> ( list ?A1 ) -> ( list ?A1 )) int) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

We can apply the expression using the rule

```
rule E1:Type E2:Type => E1 =Type (E2 -> ?T:Type) ~> ?T
```

and by pattern matching we fill in the type hole `?A1` with `int`.

```
<k>
(( list int ) -> ( list int )) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Next we need to get `model` out of the type context. By the rule

```
rule <k> variable X:Id => #renameMetaKVariables(T, Tvs) ...</k>
```

Figure 5.1.: The end result

```
<tenv>... X |-> forall Tvs . T
...</tenv>
```

we obtain the following expression.

```
<k>
(( list int ) -> ( list int )) (list ?A2)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Note how the poly type was only needed to store a record of the frozen variables. As we take a copy out of the type context, we instantiate the frozen variable resulting in a new type hole `?A1`.

Finally, we apply the expressions and again fill the type hole `?A2 = int` resulting in in our final expression.

```
<k>
list int
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Here the rewriting system terminates, and the inferred type is `list int`, as seen in Figure 5.1. Note that in the figure the type environment is empty. This is just to simplify the output.

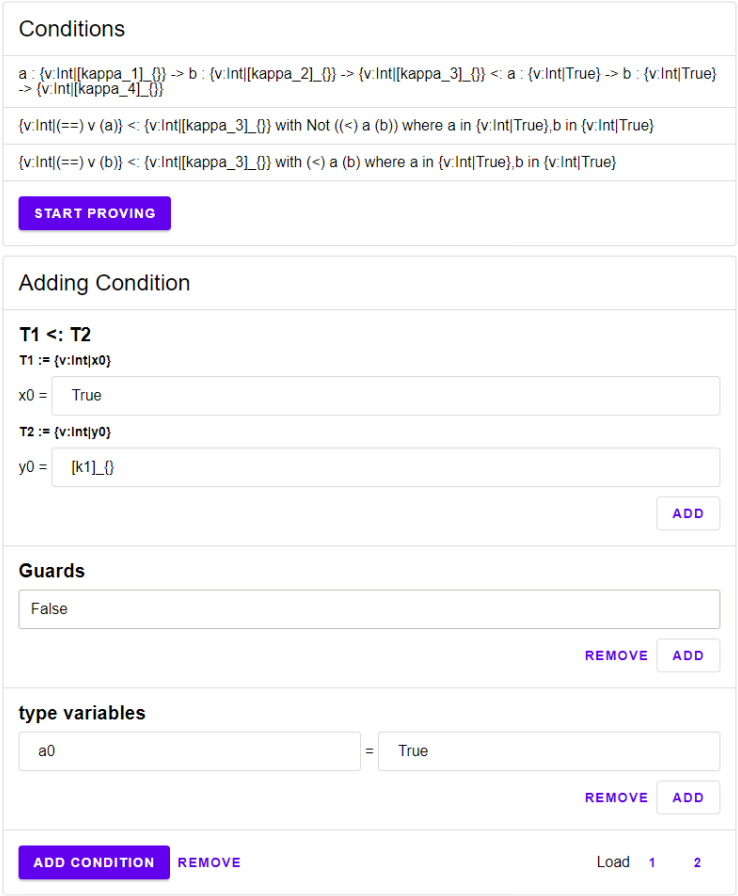


Figure 5.2.: A GUI for writing a set of input conditions.

5.2. Refinement Types in Elm

We will now turn to the implementation of the core of the type inference algorithm discussed in Section 4.4.

In particular, we will present the `split`, `solve` and `weaken` functions for computing the strongest refinements for a set of given subtyping conditions.

We have implemented these functions in Elm itself; to simplify testing, we have equipped the implementation with a GUI by using an Em package written by the author called Elm-Action [Pay20] (see Figure 5.2).

The architecture of a typical elm program is similar to that of a state machine: First a `init` function is called to define the initial state (in Elm typically called `Model`). The state is then passed to the `view` function that displays the state as an HTML document on the screen. The user can now interact with the elements on screen (like

Proof Assistant

Conditions

{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}

Partial Solution

kappa_1

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

kappa_2

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

kappa_3

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

kappa_4

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

Is the following SMT statement satisfiable? (Solve: part 1/5)

(declare-const a Int) (declare-const b Int) (declare-const v Int) (assert (and true (not (and (not (= v 0)) (and (= v 0) (and (or (< v 0) (= v 0)) (and (< v 0) (and (or (> v 0) (= v 0)) (and (> v 0) (and (not (= v b)) (and (= v b) (and (or (< v b) (= v b)) (and (< v b) (and (or (> v b) (= v b)) (and (> v b) (and (not (= v a)) (and (= v a) (and (or (< v a) (= v a)) (and (< v a) (and (or (> v a) (= v a)) (> v a))))))))))))))))))))))))))) (check-sat)

ASK SMT SOLVER auto NO YES

Figure 5.3.: Proof assistant displaying the current SMT statement

pressing a button). Once the user has performed an interaction, a message describing the action will be passed to an `update` function, updating the current state (and with that also the HTML document on screen).

Our implementation consists of three different programs called `Setup`, `Assistant` and `Done`. The `Setup` program as seen in Figure 5.2 handles the creation of our conditions. The `Assistant` program as seen in Figure 5.3 applies the `split`, `solve` and `weaken` functions to the conditions. The `Done` program as seen in Figure 5.4 shows the solution.

Our library Elm-Action simplifies the wiring to combine multiple Elm programs into one. To do so, the library models the different Elm programs as different states of a meta-level state machine: Each state is its own state machine. To transition from one program into another we define a transition function that takes some transition data as an input and returns the initial state of the new elm program.

We will only discuss the `Assistant` program, as it is the most interesting. In this program our state describes a satisfiability problem. This SAT problem needs to be solved by either the SMT solver or a human. We are using the SMT solver called Z3. To talk to Z3, we use a small JavaScript code that communicates between Z3 and Elm. Elm will send the problem in question through JavaScript to Z3 and then

Result

Conditions

{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}

Solution

kappa_1
True
kappa_2
True
kappa_3
And (Or ((>) v (b)) ((==) v (b))) (Or ((>) v (a)) ((==) v (a)))
kappa_4
And (Or ((>) v (b)) ((==) v (b))) (Or ((>) v (a)) ((==) v (a)))

Figure 5.4.: The end result

awaits a response. Once the response has been received, it will then be sent to the `update` function, resulting in a new satisfiability problem. This new problem can be again sent to either Z3 or displayed on the screen. If this process stops, then the program ends and transitions into the `Done` program.

5.3. Details of the Elm Implementation

We will now go over the Elm code in more detail.

5.3.1. Types

For Liquid Types we use the following representation:

```
type alias LiquidType a b =
    ( List
        { name : String
        , type : a
        }
    , b
    )
```

A function $a : \{Int \mid r1\} \rightarrow b : \{Int \mid r2\} \rightarrow \{Int \mid r3\}$ would be represented as `([{name=a,refinement=r1},{name=b,refinement=r2}],r3)`. We allow different types for `a` and `b`:

```
type SimpleLiquidType
    = IntType Refinement
    | LiquidTypeVariable Template
```

Possible types for `a` and `b` are either the most general `SimpleLiquidType` or the more specific types `Refinement` and `Template`. Note on the naming: `SimpleLiquidType` is "simple" in the sense that it is not a function type.

In respect to conditions we have two types:

```
type alias Condition =
    { smaller : LiquidType Template SimpleLiquidType
    , bigger : LiquidType Refinement Template
    , guards : List Refinement
    , typeVariables : List ( String, Refinement )
    }



type alias SimpleCondition =
    { smaller : SimpleLiquidType
    , bigger : Template
    , guards : List Refinement
    , typeVariables : List ( String, Refinement )
    }
```

`SimpleCondition` is the implementation of $\mathcal{C}^{-}$.

5.3.2. Transition

The `Assistant` program starts by obtaining some transition data from the `Setup` program. This transition data will then be used to initiate the state.

```
type alias Transition =
    List SimpleCondition
```

We obtain simple conditions from the *Split* function. This is a one-to-one implementation of the *Split* function previously described. We will now go through its definition.

—

```
split : Condition -> Result () (List SimpleCondition)
split =
  let
    rec : Int -> Condition -> Result () (List SimpleCondition)
    rec offset condition =
      case ( condition.smaller, condition.bigger ) of
```

```
( ( q1 :: t2, t2end ), ( q3 :: t4, t4end ) ) ->
  if q1.name == q3.name then
    rec (offset + 1)
      { condition
      | smaller = ( t2, t2end )
      , bigger = ( t4, t4end )
      , typeVariables =
        ( q3.name, q3.refinement )
          :: condition.typeVariables
      }
      |> Result.map
        ((::)
          { smaller = IntType q3.refinement
          , bigger = q1.refinement
          , guards = condition.guards
          , typeVariables = condition.typeVariables
          }
        )

  else
    Err ()
```

This first case is equivalent to the following.

$$\begin{aligned}&\mathrm{Split}(a : \{\nu : \mathit{Int} \mid q_1\} \to \hat{T}_2 <:_{\Theta,\Lambda} a : \{\nu : \mathit{Int} \mid q_3\} \to \hat{T}_4) = \\ &\quad \{\{\nu : \mathit{Int} \mid q_3\} <:_{\Theta,\Lambda} \{\nu : \mathit{Int} \mid q_1\}\} \cup \mathrm{Split}(\hat{T}_2 <:_{\Theta \cup \{(a,q_3)\},\Lambda} \hat{T}_4\})\end{aligned}$$

—

```
( ( [], q1 ), ( [], q2 ) ) ->
  [ { smaller = q1
    , bigger = q2
    , guards = condition.guards
    , typeVariables = condition.typeVariables
    }
  ]
    |> Ok
```

The second case is a direct transformation from a `Condition` into a `SimpleCondition`. For our formal definition of the second case, this is equivalent to the identity.

$$\text{Split}(\{\nu : Int \mid q_1\} <:_{\Theta,\Lambda} \{\nu : Int \mid q_2\}) =$$
$$\{\{\nu : Int \mid q_1\} <:_{\Theta,\Lambda} \{\nu : Int \mid q_2\}\}$$

—

```
        _ ->
          Err ()
  in
  rec 0
```

The *Split* function is a partial function, therefore we will return an error if neither case could be applied. If so, the `Setup` program will throw an error and the user would need to correct the given conditions. For a valid condition, the *Split* function will always be successful. Once successful the new list of `SimpleConditions` will be passed as transition data to the `Assistant` program.

```
case model.conditions |> List.map Function.split |> Result.combine of
    Ok conds ->
        conds |> List.concat |> Action.transitioning
    Err () ->
        ...
```

5.3.3. Init

After we have split the conditions, we initiate the Elm program. Note that this program will be implementing the *Solve* and *Weaken* functions.

```
init : Transition -> ( Model, Cmd Msg )
init conditions =
    let
        initList =
            (conditions
                |> List.map
                    (\{ typeVariables } ->
                        typeVariables
                            |> List.map (\( name, _ ) -> name)
                    )
                |> List.concat
            )
                |> Refinement.init
    in
```

```
    ( { conditions = conditions |> Array.fromList
      , predicates =
            conditions
                |> List.concatMap Condition.liquidTypeVariables
                |> List.map (\v -> ( v, initList |> Array.fromList ))
                |> Dict.fromList
      , index = 0
      , weaken = Nothing
      , auto = False
      , error = Nothing
      }
    , Cmd.none
    )
```

We now go through all fields of our model.

- `conditions` contains a copy of the conditions.
- `predicates` contains a dictionary, mapping every liquid type variable to the initial set of predicates $Init(V)$. (Equivalent to `Refinement.init`)
- `index` contains the index of the current condition. Keep in mind, that the loop from the *Solve* function is actually modelled as state transitions. Therefore, we can assume that we are always investigating one specific condition at a time. If not, then the program would have already stopped.
- `weaken` says if we are currently weakening a condition. If this is set to `Nothing` then we are in the *Solve* function, else its `Just i` where `i` is the index of the predicate that we are currently investigating.
- `auto` is a boolean expression that says if the SMT solver should be asked directly. If set to `False`, then the user may decide the satisfiability of the current SMT statement.
- `error` contains any error message that should be displayed to the user. These errors come directly from the SMT solver.

5.3.4. Update

```
update : (String -> Cmd msg) -> Msg -> Model -> Update msg
update sendMsg msg model =
    case msg of
        GotResponse bool ->
            handleResponse sendMsg bool { model | error = Nothing }
        ...
```

```
handleResponse : (String -> Cmd msg) -> Bool -> Model -> Update msg
handleResponse sendMsg bool model =
    case model.weaken of
        Just weaken ->
            handleWeaken weaken sendMsg bool model

        Nothing ->
            handleSolve sendMsg bool model
```

We have stored the additional information needed for the *Weaken* function in `model.weaken`. We therefore check the content of `model.weaken`. We check the content of `model.weaken`, If it is `Nothing` we know that we are in the *Solve* function, else we know that we are currently in the *Weaken* function.

5.3.4.1. The Solve Function

```
handleSolve : (String -> Cmd msg) -> Bool -> Model -> Update msg
handleSolve sendMsg bool model =
    if bool then
        --Start weaking
        case
            model.conditions
                |> Array.get model.index
        of
            Just { bigger } ->
                { model
                    | weaken =
                        Just
                            { index = 0
                            , liquidTypeVariable =
                                bigger |> Tuple.first
                            }
                }
                    |> handleAuto sendMsg

            Nothing ->
                Action.updating ( model, Cmd.none )
```

If the incoming result is `True` it means that the SMT statement is satisfiable. Therefore, we start the *Weaken* function. To do so, we initiate the weakening index at `0` and also store the liquid type variable whose corresponding refinement we want to weaken.

—

```
    else
        --Continue
        let
            index =
                model.index + 1
        in
        if index >= (model.conditions |> Array.length) then
            Action.transitioning
                { conditions = model.conditions
                , predicates =
                    model.predicates
                        |> Dict.map
                          (\_ -> Array.toList >> Refinement.conjunction)
                }

        else
            { model
                | index = index
            }
                |> handleAuto sendMsg
```

If the incoming result is `False`, then we check out the next condition. If there exists no following condition, then the function is done. We end the Elm program by transitioning into the `Done` program.

5.3.4.2. The Weaken Function

```
handleWeaken :
    { index : Int
    , liquidTypeVariable : Int
    }
    -> (String -> Cmd msg)
    -> Bool
    -> Model
```

```
    -> Update msg
handleWeaken weaken sendMsg bool model =
    if bool then
        --Remove
        let
            predicates =
                model.predicates
                    |> Dict.update weaken.liquidTypeVariable
                        (Maybe.map
                            (Array.removeAt weaken.index)
                        )
        in
        if
            weaken.index
                >= (predicates
                        |> Dict.get weaken.liquidTypeVariable
                        |> Maybe.map Array.length
                        |> Maybe.withDefault 0
                   )
        then
            { model
                | predicates = predicates
                , weaken = Nothing
                , index = 0
            }
                |> handleAuto sendMsg

        else
            { model
                | predicates = predicates
            }
                |> handleAuto sendMsg
```

If the incoming result is `False`, then the SMT statement is unsatifiable. Thus, we remove the predicate. If no predicate exists, we finish the *Weaken* function by setting `model.weaken` to `Nothing`.

—

```
    else
        --Continue
```

```
let
    index =
        weaken.index + 1
in
if
    index
        >= (model.predicates
                |> Dict.get weaken.liquidTypeVariable
                |> Maybe.map Array.length
                |> Maybe.withDefault 0
           )
then
    { model
        | weaken = Nothing
        , index = 0
    }
        |> handleAuto sendMsg

else
    { model
        | weaken =
            Just
                { liquidTypeVariable = weaken.liquidTypeVariable
                , index = index
                }
    }
        |> handleAuto sendMsg
```

If the incoming result is `True`, then the SMT statement is satisfiable. We therefore check out the next predicate. We finish the function if no following predicate exists. To do so we again set `model.weaken` to `Nothing`.

5.3.5. The SMT Statement

After every update we check if the SMT statement should be automatically sent to the SMT solver.

```
handleAuto : (String -> Cmd msg) -> Model -> Update msg
handleAuto sendMsg model =
    if model.auto then
```

```
        ( model
        , model
            |> smtStatement
            |> Maybe.map sendMsg
            |> Maybe.withDefault Cmd.none
        )
            |> Action.updating

    else
        Action.updating
            ( model, Cmd.none )
```

If not, it will be displayed on the screen. Either way we need to compute the SMT statement for the given model.

```
smtStatement : Model -> Maybe String
smtStatement model =
    let
        toString : SimpleCondition -> String
        toString condition =
            case model.weaken of
                Just weaken ->
                    statementForWeaken weaken model condition

                Nothing ->
                    statementForSolve model condition
    in
    model.conditions
        |> Array.get model.index
        |> Maybe.map toString
```

The statement differes between the *Solve* and the *Weaken* function.

5.3.5.1. The SMT Statement for Solve

For the *Solve* function we translate the condition directly into the SMT statement.

```
statementForSolve : Model -> SimpleCondition -> String
statementForSolve model condition =
    condition
        |> Condition.toSMTStatement
            (model.predicates
```

```
                |> Dict.map (\_ -> Array.toList
                  >> Refinement.conjunction)
            )
```

The actual translation happens in `Condition.toSMTStatement`. The translation is taken directly from the described *Solve* function. We therefore will now compare both with another.

—

```
toSMTStatement : Dict Int Refinement -> SimpleCondition -> String
toSMTStatement dict { smaller, bigger, guards, typeVariables } =
    let
        typeVariablesRefinements : List Refinement
        typeVariablesRefinements =
            typeVariables
                |> List.map
                    (\( b, r ) ->
                        r |> Refinement.rename
                            { find = "v"
                            , replaceWith = b
                            }
                    )
```

This is equivalent to the following.

$$
\begin{aligned}
&\text{Let}\\
&\Theta' := \{\ (a, r)\\
&\qquad \mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q}\\
&\qquad \vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta\\
&\qquad\qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\}\\
&\{(b_1, r'_1), \ldots, (b_n, r'_n)\} = \Theta'\\
&\text{in } \bigwedge_{j=0}^{n} [r'_j]_{\{(\nu, b_j)\}}
\end{aligned}
$$

—

```
        r1 : Refinement
        r1 =
            case smaller of
                IntType refinement ->
```

```
        refinement

    LiquidTypeVariable ( int, list ) ->
        list
            |> List.foldl
                (\( k, v ) ->
                    Refinement.substitute
                        { find = k
                        , replaceWith = v
                        }
                )
                (dict
                    |> Dict.get int
                    |> Maybe.withDefault IsFalse
                )
```

Here we have a case distinction between a refinement and a liquid type variable. We had the same distinction in our original definition of $r1$:

$$r_1 := \begin{cases} \bigwedge [S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \text{ and } S_1 \in \mathcal{V} \nrightarrow \mathit{IntExp} \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases},$$

—

```
r2 : Refinement
r2 =
    bigger
        |> Tuple.second
        |> List.foldl
            (\( k, v ) ->
                Refinement.substitute
                    { find = k
                    , replaceWith = v
                    }
            )
            (dict
                |> Dict.get (bigger |> Tuple.first)
                |> Maybe.withDefault IsFalse
            )
```

Here we see how we apply the lazy substitution (stored in `bigger |> Tuple.second`). In the original definition, we assumed that we know how to apply a substitution on the term level:

$$r_2 := \bigwedge [S(\kappa_2)]_{S_2}$$

—

```
        statement : Refinement
        statement =
            (r1
                :: typeVariablesRefinements
                ++ guards
            )
                |> List.foldl AndAlso (IsNot r2)
    in
    (statement
        |> Refinement.variables
        |> Set.toList
        |> List.map (\k -> "(declare-const " ++ k ++ " Int)\n")
        |> String.concat
    )
        ++ ("(assert "
               ++ (statement |> Refinement.toSMTStatement)
               ++ ")\n(check-sat)"
           )
```

The final statement is therefore

$$((\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu, b_j)\}}) \wedge r_1 \wedge p) \wedge \neg r_2$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

5.3.5.2. The SMT Statement for Weaken

For the *Weaken* function we modify the statement.

```
statementForWeaken :
    { index : Int, liquidTypeVariable : Int }
    -> Model
```

```
    -> SimpleCondition
    -> String
statementForWeaken weaken model condition =
  condition
    |> Condition.toSMTStatement
      (model.predicates
        |> Dict.map (\_ -> Array.toList >> Refinement.conjunction)
        |> Dict.update (condition.bigger |> Tuple.first)
          (Maybe.map
            (\_ ->
              model
                |> getLazySubstitute
                |> List.foldl
                  (\( find, replaceWith ) ->
                    Refinement.substitute
                      { find = find
                      , replaceWith = replaceWith
                      }
                  )
                  (model.predicates
                    |> Dict.get (condition.bigger |> Tuple.first)
                    |> Maybe.andThen (Array.get weaken.index)
                    |> Maybe.withDefault IsFalse
                  )
            )
          )
      )
```

We replace the value at the point `condition.bigger |> Tuple.first` with the predicate in question. The same happens in our formal definition. The resulting SMT statement for the predicate q is therefore

$$((\bigwedge_{j=0}^{n} [r'_j]_{\{(\nu, b_j)\}}) \wedge r_1 \wedge p) \wedge \neg q$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

We therefore swap the result around: We keep the predicate if we SMT statement is unsatifiable. This is equivalent to saying we keep the predicate if the negated SMT statement is satifiable:

$$\neg((\bigwedge_{j=0}^{n}[r'_j]_{\{(\nu,b_j)\}}) \wedge r_1 \wedge p) \vee q$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

5.4. Demonstration

For a demonstration, we will consider the following function.

```
max : a:{ v:Int|True } -> b:{ v:Int|True } -> { v:Int|k4 };
max =
  \a -> \b ->
    if
      (<) a b
    then
      b
    else
      a
```

To check the validity of the type signature, we will first infer the type of the function and then compare it with the type signature. Using the inference rule, we obtain as a result the type

$$\{v : Int \mid \kappa_1\} \rightarrow \{v : Int \mid \kappa_2\} \rightarrow \{v : Int \mid \kappa_3\}$$

with the following conditions.

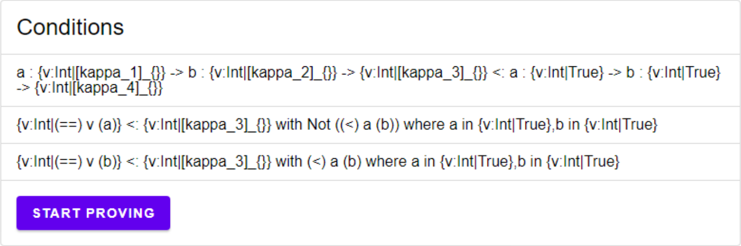
$$\{\nu : Int \mid \nu = b\} <:_{\{(a,\{Int \mid True\}),(b,\{Int \mid True\})\},\{a<b\}} \{\nu : Int \mid \kappa_3\},$$
$$\{\nu : Int \mid \nu = a\} <:_{\{(a,\{Int \mid True\}),(b,\{Int \mid True\})\},\{\neg(a<b)\}} \{\nu : Int \mid \kappa_3\},$$

We now write the validity check of the type signature as a condition.

$$a : \{\nu : Int \mid \kappa_1\} \rightarrow b : \{\nu : Int \mid \kappa_2\} \rightarrow \{\nu : Int \mid \kappa_3\}$$
$$<:_{\{\},\{\}} a : \{\nu : Int \mid True\} \rightarrow b : \{\nu : Int \mid True\} \rightarrow \{\nu : Int \mid \kappa_4\}$$

Figure 5.5 shows how the conditions can be inserted into the elm program.

If we click on the "Start Proving" button, the `Assistant` program will start and get the list of conditions as the transition data. It now applies the *Split* function to the conditions and computes the first SMT statement, as seen in Figure 5.6.

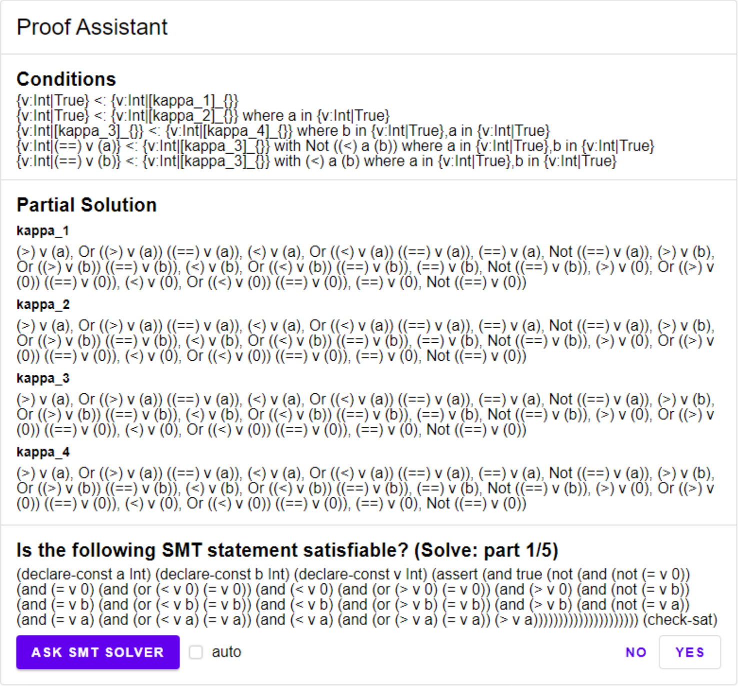

Conditions

a : {v:Int|[kappa_1]_{}} -> b : {v:Int|[kappa_2]_{}} -> {v:Int|[kappa_3]_{}} <: a : {v:Int|True} -> b : {v:Int|True} -> {v:Int|[kappa_4]_{}}

{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}

{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}

Figure 5.5.: The conditions of the max-function


Proof Assistant

Conditions

{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}

Partial Solution

kappa_1

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

kappa_2

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

kappa_3

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

kappa_4

(>) v (a), Or ((>) v (a)) ((==) v (a)), (<) v (a), Or ((<) v (a)) ((==) v (a)), (==) v (a), Not ((==) v (a)), (>) v (b), Or ((>) v (b)) ((==) v (b)), (<) v (b), Or ((<) v (b)) ((==) v (b)), (==) v (b), Not ((==) v (b)), (>) v (0), Or ((>) v (0)) ((==) v (0)), (<) v (0), Or ((<) v (0)) ((==) v (0)), (==) v (0), Not ((==) v (0))

Is the following SMT statement satisfiable? (Solve: part 1/5)

(declare-const a Int) (declare-const b Int) (declare-const v Int) (assert (and true (not (and (not (= v 0)) (and (= v 0) (and (or (< v 0) (= v 0)) (and (< v 0) (and (or (> v 0) (= v 0)) (and (> v 0) (and (not (= v b)) (and (= v b) (and (or (< v b) (= v b)) (and (< v b) (and (or (> v b) (= v b)) (and (> v b) (and (not (= v a)) (and (= v a) (and (or (< v a) (= v a)) (and (< v a) (and (or (> v a) (= v a)) (> v a))))))))))))))))))))) (check-sat)

Figure 5.6.: The conditions of the max-function

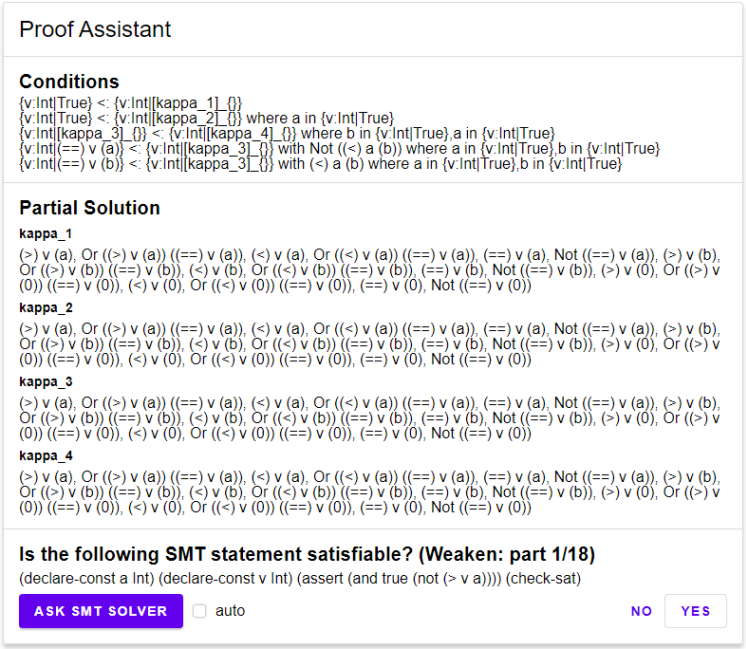


Figure 5.7.: Weakening the predicates.

Here we see the conditions on top, displaying the conditions that are now split. Next we see that for each `kappa` the set of predicated have been initiated with all possible predicates for variables `a` and `b`. Below it presents the first SMT statement in the `Solve`-step, mainly if the first condition is not satisfiable for the current value of `kappa_1`. Therefore, the SMT statement is satisfiable.

Next, the program goes into *Weaken*-mode and starts checking each and every predicate currently associated with `kappa_1`, as seen in Figure 5.7.

Once it has checked every predicate, it goes back to *Solve*-mode and repeats. In Figure 5.8 one can see the result after a few iterations.

Once every condition is valid (meaning that all SMT statements in the *Solve*-mode are unsatifiable) the program holds and the result is displayed as seen in Figure 5.4.

Proof Assistant

Conditions

{v:Int|True} <: {v:Int|[kappa_1]_{}}
{v:Int|True} <: {v:Int|[kappa_2]_{}} where a in {v:Int|True}
{v:Int|[kappa_3]_{}} <: {v:Int|[kappa_4]_{}} where b in {v:Int|True},a in {v:Int|True}
{v:Int|(==) v (a)} <: {v:Int|[kappa_3]_{}} with Not ((<) a (b)) where a in {v:Int|True},b in {v:Int|True}
{v:Int|(==) v (b)} <: {v:Int|[kappa_3]_{}} with (<) a (b) where a in {v:Int|True},b in {v:Int|True}

Partial Solution

kappa_1

kappa_2

kappa_3

Or ((>) v (a)) ((==) v (a)), Or ((<) v (a)) ((==) v (a)), (==) v (a), Or ((>) v (b)) ((==) v (b))

kappa_4

Or ((>) v (a)) ((==) v (a)), Or ((<) v (a)) ((==) v (a)), (==) v (a), Or ((>) v (b)) ((==) v (b))

Is the following SMT statement satisfiable? (Solve: part 1/5)

(assert (and true (not true))) (check-sat)

ASK SMT SOLVER auto NO YES

Error : WARNING: input file was already specified.

Figure 5.8.: The partial result after a few iterations.

6. Conclusions

In this thesis, we have investigated the type system for the Elm language and discussed its extension by refinement types.

The original intent was to have an implementation of the type checker for liquid types. We expected that the resulting liquid types are defined such that non-negative integers, range types and non-zero integers can be defined. We expected to implement liquid types for `Int`, `Bool` and tuples of liquid types. Additionally, we expected that the inferred type of the `max` function can be sharp. Such a sharp refinement is $(a \leq \nu) \wedge (b \leq \nu) \wedge (\nu = b \vee \nu = a)$.

Indeed, the resulting type system is capable of defining all described integer types and also allows inferring liquid types for functions over integers. However, It does not include liquid types over `Bool` and tuples. Additionally, the inferred type of the max function is not sharp: $(a \leq \nu) \wedge (b \leq \nu)$. Adding these missing features would have been too time-consuming and would not provide any new revelations:

- The inclusion of Booleans would have meant that we would have needed to type check the liquid expressions. This would have added unnecessary complexity, as we are mainly interested in subtypes of integers.
- Tuples would have been easy to add but would not yield additional behaviour, as Tuples as arguments can be flattened and then transformed into a list of arguments and Tuples as a return argument are syntax sugar for defining multiple functions with the same input values.
- To infer a sharp refinement for the `max` function, one can simply add $\nu = b \vee \nu = a$ to the search space, but that is not very sophisticated. Another way would be to add $P \vee Q$ for a specific set of allowed predicates for P and Q. We used our Elm implementation to quickly test this for the definition of P and Q not containing $\vee$. The resulting refinement was sharp but included a lot of trivial conditions. The search space increased by a factor of four. This factor could be decreased by ensuring that no two predicates are equivalent.

While working on the thesis it became clear that the original expectations did not completely match the possibilities of liquid types. In particular, the expressiveness of liquid types is directly dependent on the initial set of predicates and the allowed

expressions in $\mathcal{Q}$. Extending the allowed expressions in $\mathcal{Q}$ requires that the SMT solver can still cope with them. In contrast to our original expectation, the set of predicates $\mathcal{Q}$ allowed in if-branches is a superset of the predicates allowed in refinements. Additionally, the search space for the derived predicates must be finite. This means that no matter how big the space we are considering is, there will always be a predicate in $\mathcal{Q}$ that cannot be found.

There are multiple future topics that can be explored.

- The set of allowed expressions $\mathcal{Q}$ and the search space for the inferred refinements can be extended to sharpen the inferred predicates. At some point, this would also need to include an algorithm to simplify the inferred predicates. Otherwise, the inferred predicates can only be hardly read by humans.
- The current implementation in Elm can be extended to a full type checker by using the Elm-in-Elm compiler [Jan19]. This would require some changes to the type checker part of the Elm-in-Elm compiler. The updated checker would need to collect the subtyping conditions while inferring the type (as discussed in Section 4). This can not be done by simply traversing the abstract syntax tree. Such an addition would be simple but tedious, as every type inference rule would need to be updated.
- One can try to implement a specific type in Elm without liquid types. Liquid types make a type system incomplete. Therefore, this is a far better solution of implementing a specific type. For the range type, the author of this thesis has actually found another way, namely to implement these types using phantom types (an algebraic type were not all type variables are used) [Pay21].

Working on this thesis gave insights into the inner workings of liquid types. It showcased its strengths but also its weaknesses. Liquid types are an interesting topic but are not really a good fit for the Elm language and the philosophies behind it.

A. Source Code

The source code discussed in this thesis can be found under

`https://github.com/orasund/elm-refine`.

Bibliography

[Bac59] John W. Backus. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". In: *IFIP Congress*. 1959, pp. 125–131.

[Ben+08] Jesper Bengtson et al. "Refinement Types for Secure Implementations". In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. 2008, pp. 17–32. DOI: 10.1109/CSF.2008.27. URL: https://doi.org/10.1109/CSF.2008.27.

[BR15] Denis Bogdanas and Grigore Roşu. "K-Java: A Complete Semantics of Java". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 445–456. ISSN: 0362-1340. DOI: 10.1145/2775051.2676982. URL: https://doi.org/10.1145/2775051.2676982.

[CC13] Evan Czaplicki and Stephen Chong. "Asynchronous functional reactive programming for GUIs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 411–422. DOI: 10.1145/2491956.2462161. URL: https://doi.org/10.1145/2491956.2462161.

[Chr18] Daniel P. Friedman; David Thrane Christiansen. *The Little Typer*. Paperback. The MIT Press, 2018. ISBN: 0262536439,9780262536431.

[Cza21] Evan Czaplicki. *Elm Website*. 2021. URL: https://elm-lang.org/.

[DM82] Luís Damas and Robin Milner. "Principal Type-Schemes for Functional Programs". In: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*. 1982, pp. 207–212. DOI: 10.1145/582153.582176. URL: https://doi.org/10.1145/582153.582176.

[FM14] Daniele Filaretti and Sergio Maffeis. "An Executable Formal Semantics of PHP". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 567–592. ISBN: 978-3-662-44202-9.

[FP91] Timothy S. Freeman and Frank Pfenning. "Refinement Types for ML". In: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. 1991, pp. 268–277. DOI: 10.1145/113445.113468. URL: https://doi.org/10.1145/113445.113468.

[Fre84] Gottlob Frege. *Die Grundlagen der Arithmetik – Eine logisch mathematische Untersuchung über den Brgiff der Zahl*. Breslau: Verlag von Wilhelm Koebner, 1884. URL: http://www.gutenberg.org/ebooks/48312,%20https://archive.org/details/diegrundlagende01freggoog.

[Gut+16] Dwight Guth et al. "RV-Match: Practical Semantics-Based Program Analysis". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Vol. 9779. LNCS. Springer, July 2016, pp. 447–453. DOI: http://dx.doi.org/10.1007/978-3-319-41528-4_24.

[Gut13] Dwight Guth. "A formal semantics of Python 3.3". In: 2013. URL: https://github.com/kframework/python-semantics.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the Undefinedness of C". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 336–345. ISSN: 0362-1340. DOI: 10.1145/2813885.2737979. URL: https://doi.org/10.1145/2813885.2737979.

[Hil+18] Everett Hildenbrandt et al. "KEVM: A Complete Semantics of the Ethereum Virtual Machine". In: *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.

[Jan19] Martin Janiczek. *Elm-In-Elm*. 2019. URL: https://github.com/elm-in-elm/compiler.

[Kan+18] Shuanglong Kan et al. "K-Rust: An Executable Formal Semantics for Rust". In: *CoRR* abs/1804.07608 (2018). arXiv: 1804.07608. URL: http://arxiv.org/abs/1804.07608.

[Kaz+18] Milod Kazerounian et al. "Refinement Types for Ruby". In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*. 2018, pp. 269–290. DOI: 10.1007/978-3-319-73721-8_13. URL: https://doi.org/10.1007/978-3-319-73721-8%5C_13.

[KKT16] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. "Occurrence typing modulo theories". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 2016, pp. 296–309.

DOI: 10.1145/2908080.2908091. URL: https://doi.org/10.1145/2908080.2908091.

[KLN04] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory: From its Origins until Today*. Applied Logic 29. Kluwer, 2004.

[KRJ10] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. "Dsolve: Safety Verification via Liquid Types". In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 2010, pp. 123–126. DOI: 10.1007/978-3-642-14295-6_12. URL: https://doi.org/10.1007/978-3-642-14295-6%5C_12.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems* 4963 (Apr. 2008), pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[Mil78] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[Par19] Matt Parker. *Humble Pi:A Comedy of Maths Errors*. ISBN 978-0241360194. Allen Lane, 2019.

[Pay20] Lucas Payr. *Elm-Action*. 2020. URL: https://github.com/Orasund/elm-action.

[Pay21] Lucas Payr. *Elm-Static-Array*. 2021. URL: https://github.com/Orasund/elm-static-array.

[Pea89] G. Peano. *Arithmetices principia: nova methodo*. Trans. by Vincent Verheyen. Fratres Bocca, 1889. URL: https://github.com/mdnahas/Peano_Book/blob/master/Peano.pdf.

[Pie+02] B.C. Pierce et al. *Types and Programming Languages*. The MIT Press. MIT Press, 2002. ISBN: 9780262162098. URL: https://books.google.at/books?id=ti6zoAC9Ph8C.

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN: 0262162288.

[PSR15] Daejun Park, Andrei Stefănescu, and Grigore Roşu. "KJS: A Complete Formal Semantics of JavaScript". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 346–356. ISSN: 0362-1340. DOI: 10.1145/2813885.2737991. URL: https://doi.org/10.1145/2813885.2737991.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Liquid types". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 2008, pp. 159–169. DOI: 10.1145/1375581.1375602. URL: https://doi.org/10.1145/1375581.1375602.

[RKJ10] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Low-level Liquid types". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010, pp. 131–144. DOI: 10.1145/1706299.1706316. URL: https://doi.org/10.1145/1706299.1706316.

[Ron+12] Patrick Maxim Rondon et al. "CSolve: Verifying C with Liquid Types". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 2012, pp. 744–750. DOI: 10.1007/978-3-642-31424-7_59. URL: https://doi.org/10.1007/978-3-642-31424-7%5C_59.

[RS14] Grigore Rosu and Traian-Florin Serbanuta. "K Overview and SIMPLE Case Study". In: *Electr. Notes Theor. Comput. Sci.* 304 (2014), pp. 3–56. DOI: 10.1016/j.entcs.2014.05.002. URL: https://doi.org/10.1016/j.entcs.2014.05.002.

[Ste+16] Andrei Stefănescu et al. "Semantics-Based Program Verifiers for All Languages". In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 74–91. ISSN: 0362-1340. DOI: 10.1145/3022671.2984027. URL: https://doi.org/10.1145/3022671.2984027.

[Vaz+14] Niki Vazou et al. "Refinement types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 2014, pp. 269–282. DOI: 10.1145/2628136.2628161. URL: https://doi.org/10.1145/2628136.2628161.

[VCJ15] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. "Trust, but Verify: Two-Phase Typing for Dynamic Languages". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 2015, pp. 52–75. DOI: 10.4230/LIPIcs.ECOOP.2015.52. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2015.52.

[VCJ16] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. "Refinement types for TypeScript". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*

2016, Santa Barbara, CA, USA, June 13-17, 2016. 2016, pp. 310–325. DOI: 10.1145/2908080.2908110. URL: https://doi.org/10.1145/2908080.2908110.

[VRJ13] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. "Abstract Refinement Types". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 209–228. DOI: 10.1007/978-3-642-37036-6_13. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_13.

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.