# Liquid Type Inference Under the Hood

**Micha Reiser**

University of Applied Sciences Rapperswil
Supervised by Prof. Dr. Farhad D. Mehta
Seminar AT 2016

Dependent types can be used to prove more fine-granular invariants of programs. However, dependent types come at the cost of requiring to be manually annotated. Since dependent types are awkward to write, it is desired to have automatically inferred dependent types. The liquid type inference algorithm explained in this paper is capable of inferring dependent types that only use conjunctions over a given set of qualifiers. The constraint-based inference algorithm uses Hindley-Milner to infer the ML-types and check if the program is well-typed in the sense of the underlying type system. This guarantee reduces dependent types inference to inferring the unknown refinement predicates. The algorithm consists of three steps: Firstly, create templates for the unknown refinement predicates. Secondly, generate constraints over these templates setting them into a relation. Finally, find the least fixpoint solution for the unknown refinement predicates that satisfies all the constraints. The program is well-typed if the least fixpoint solution can be found. This paper further shows why path sensitivity is an important property of liquid type inference and how it is realized.

## 1 Introduction

Static type checking is a program verification technique used in many programming languages to identify "erroneous terms" at compile time. However, the kind of errors identifiable by a type checker depends on the granularity of the type system. Most of the classical type systems are coarse-grained, in a way that, a variable `x = 0` can be defined to be an instance of the type `int` but not that its value is in a certain range or, more precisely, is equal to zero. This loss of information requires the compiler to add additional runtime checks to guarantee the safety of programs at runtime — e.g. by checking that a divisor is not equal to zero.

Classical type systems are also too coarse-grained to detect if an index based list or array access potentially is out of range and therefore, results in an "index out of range error" at runtime. For example, the `head` function of Haskell returns the first element of a non-empty list but fails at runtime if the list is empty. Therefore, a type system is desired capable of expressing that the argument of the *head* function has to be a non-empty list.

*Dependent types*, denoted as $T$, address this lack of expressibility by adding invariants to types describing the values a type may hold [1]. They allow refining the set of valid values for a structural type (e.g. **int**).

**Definition 1 (Dependent Type $T$).** *A dependent type is a type with an invariant that restricts the set of valid values. The invariant may refer to values of other types.*

A dependent type in its general form is defined in (1) in which $B$ defines the base type, the structure of the values.

$$\{\nu : B \mid p\} \tag{1}$$

**Definition 2 (Base Type $B$).** *The base type defines the structure of the values of an expression. Examples of base types are **int**, **char**, and **bool**.*

The *refinement predicate p* is a logical expression that must hold for all valid values of the dependent type. The base type of the variable `x` can be refined using the refinement predicate $\nu = 0$ stating that the only valid value is the value zero.

**Definition 3 (Refinement Predicate $p$).** *The refinement predicate p is a logical predicate refining (restricting) the set of valid values of a base type for the dependent type.*

The refinement predicate references the special *value variable ν*. The value variable acts as a placeholder for a value tested if it is part of the dependent type or not. The refinement predicate must evaluate to true if $\nu$ is replaced with a valid value of the dependent type and otherwise to false. The refinement predicate for the variable x from the example $x = 0$ evaluates to true if $\nu$ is replaced with zero ($0 = 0$) and to false for all values unequal to zero ($1 = 0$). The most precise dependent type for the variable $x$ is $\{\nu : \textbf{int} \mid \nu = 0\}$.

**Definition 4 (Value Variable $\nu$).** *The value variable $\nu$ is a placeholder in the refinement predicate that — if replaced* *with a value of the base type — states if the value is part of this dependent type.*

The special refinement predicate true is valid for all values and is, therefore, no real refinement over the base type. On the other hand, the refinement predicate false excludes all values from the base type, leaving the empty set for the valid values of the dependent type.

The expressibility of dependent types allows the compiler to prove further invariants and omitting no longer needed runtime checks. However, dependent types have the disadvantage that type inference — inferring the dependent type for every expression in a program without the need for explicit type annotations — is undecidable. This limitation adds the burden of manually annotating the dependent types to the programmer [2]. The manual annotating is undesired, because as Pierce stated:

> The more interesting your types get, the less fun it is to write them down [3].

*Liquid types* (an abbreviation for Logical Qualified Data Types) addresses this issue by providing a system capable of inferring a subset of dependent types. Liquid types have the same structure as dependent types but differ in the allowed refinement predicates. The refinement predicate has to be a logical conjunction (e.g. $p_1 \wedge p_2$) over the set of qualifiers $\mathbb{Q}^\star$.

**Definition 5 (Liquid Type $\hat{T}$).** *A liquid type $\hat{T}$ is a dependent type where the refinement predicate is a conjunction over the qualifiers from $\mathbb{Q}^\star$.*

The notation for liquid types is $\hat{T}_{\mathbb{Q}}$ in which $\mathbb{Q}$ defines the usable qualifiers. The definition of the qualifiers $\mathbb{Q}$ in this paper is:

$$\mathbb{Q} = \{0 \le v, v \ne 0, \star \le v, v < \star\} \tag{2}$$

A valid refinement predicate for a liquid type with the runtime value two over $\mathbb{Q}$ is $0 \le v \wedge v \ne 0$. The liquid type can be abbreviated as $\hat{T}$ if $\mathbb{Q}$ is evident from the context.

**Definition 6 (Qualifiers $\mathbb{Q}$).** *The set of qualifiers $\mathbb{Q}$ defines the usable qualifiers for refinement predicates p of liquid types.*

The qualifiers in $\mathbb{Q}$ may use the value variable $v$ as well as the special $\star$-*variable*. The $\star$-variable is a placeholder for any program variable. The qualifiers containing $\star$-variables can be *instantiated* by replacing the $\star$-variable with an actual variable from the program context. The qualifiers $\mathbb{Q}^\star$ for a program with the variables $x$ and $y$ are the ones defined in (3). The difference to $\mathbb{Q}$ is that the qualifiers containing the $\star$-variable are instantiated with the program variables $x$ and $y$.

$$\mathbb{Q}^\star = \{0 \le v, v \ne 0, x \le v, y \le v, v < x, v < y\} \tag{3}$$

**Definition 7 ($\mathbb{Q}^\star$).** *The set of qualifiers $\mathbb{Q}^\star$ matches the qualifiers of $\mathbb{Q}$ where all $\star$-variables are instantiated with variables from the program context.*

The set of qualifiers $\mathbb{Q}$ influences the precision of the inferred dependent types and the provable invariants. With the current set of qualifiers, the invariant requiring that the `head` function from Haskell be never called with an empty list can not be proven. The additional qualifier $len \star \ne 0$ is needed measuring if the length of a list is unequal to zero. If $\mathbb{Q}$ is extended with this new qualifier, then the invariant can be proven by using the refinement predicate $len\ list \ne 0$ for the $list$ argument.

The type inference algorithm infers the base type $B$ as well as the refinement predicate $p$ for all expressions in the program if it is well-typed. It fails if the program is ill-typed. The algorithm supports parametric polymorphism [4] as well as recursion — constructs that are problematic to cover with data-flow-analysis. The programmer can manually annotate the dependent type if the inferred dependent type is not precise enough.

The goal of the following sections is to explain how liquid types can be used to infer dependent types and verify if a program is well-typed or not. Section 2 introduces the language L1 — together with the type inference rules — that is used to define the $\phi$-function. The $\phi$-function is used in section 3 to explain the type inference algorithm step by step. The first step of the algorithm is to create templates for the unknown refinement predicates as described in section 3.1. The following section 3.2 shows how the constraints for the created templates are extracted from the program. These constraints are solved in section 3.3. The section 3.4 highlights why the algorithm needs to be path-sensitive to be useful and how path sensitivity is accomplished. Section 4 summarizes the required steps.

## 2 Language L1

This section introduces the language L1 which is used in the examples throughout the paper. This section also defines the type inference rules utilized by the liquid type inference algorithm. However, the explanation of the rules is deferred up to the point where the rules are first used.

The figure 1 shows the syntax of the language L1. The language is a simply typed lambda calculus based language but differs in the notation for function abstractions. The chosen notation is inspired by the lambda-notation in c# or JavaScript with the intent to make it more readable for persons less familiar with the lambda calculus. The figure 1 also defines the syntax for the liquid refinement predicates $Q$ that are either a single qualifier or a conjunction over the qualifiers from $\mathbb{Q}^\star$.

**Remark 1.** *Even though liquid type inference supports parametric polymorphism the language L1 does not for brevity. Support for parametric polymorphism can be added by extending the syntax with the definition for type variables, type schemas, polytypes, type abstractions and type instantiations.*

*Expressions*

| | |
|---|---|
| $e ::=$ | *Expressions:* |
| $x$ | variable |
| $\mid c$ | constant |
| $\mid x \Rightarrow e$ | abstraction |
| $\mid e\ e$ | application |
| $\mid$ **let** $x = e$ **in** $e$ | let-binding |
| $\mid$ **if** $e$ **then** $e$ **else** $e$ | if-then-else |
| $\mid e$ op $e$ | binary-operation |
| $op ::=$ | *Operations* |
| $\mid +$ | plus-operator |
| $\mid /$ | division-operator |
| $\mid =$ | equality-operator |
| $\mid <$ | less-than-operator |
| $\mid >$ | greater-than-operator |
| $Q ::=$ | *Liquid Refinements* |
| true | true |
| $\mid$ false | false |
| $\mid q$ | logical qualifier in $\mathbb{Q}^\star$ |
| $\mid Q \wedge Q$ | conjunction of qualifiers |

Fig. 1. Syntax of the Language L1

*The complete syntax and the additional required type inference rules can be taken from the original Liquid Types paper [5].*

Besides the language, a clear notion of the different kind of types is needed. The types are defined in figure 2. The base types $B$ are the types for the primitive values supported by the language. The ML-types — denoted as $\tau$ — are the not refined types defining the shape of a value and are either a base or a function type. A function type is composed out of two ML-types where the first type is the type of the argument and the second the type of the returned value. These definitions are common for classical type systems.

Dependent types — denoted as $T$ — are composed of a base type and a refinement predicate over the qualifiers $E$. The set of qualifiers $E$ is not further specified since liquid type inference does not depend on its definition. However, $E$ needs to be a superset of $\mathbb{Q}^\star$ to guarantee that the inferred liquid types are sound dependent types. Dependent functions are structured equally as ML-function types with the difference that the argument and return types are dependent types as well. Liquid types — denoted using $\hat{T}$ — are defined similarly as dependent types with the difference that the qualifiers are over $\mathbb{Q}^\star$ instead of $E$.

The type environment is a mapping from program variables to dependent types and is denoted as $\Gamma$.

| | |
|---|---|
| $B ::=$ | *Base Types:* |
| **int** | base type of integers |
| **bool** | base type of booleans |
| $\tau ::=$ | *ML-Types:* |
| $B$ | base type |
| $x : \tau \rightarrow \tau$ | function |
| $T ::=$ | *Dependent Types* |
| $\{\nu : B \mid E\}$ | dependent base type |
| $x : T \rightarrow T$ | dependent function |
| $\hat{T} ::=$ | *Liquid Types*: |
| $\{\nu : B \mid Q\}$ | liquid base type |
| $x : \hat{T} \rightarrow \hat{T}$ | liquid function |

Fig. 2. Types of the Language L1

$$\begin{aligned}
\text{true} &:: \{\nu : \textbf{bool} \mid \nu\} \\
\text{false} &:: \{\nu : \textbf{bool} \mid \neg\nu\} \\
3 &:: \{\nu : \textbf{int} \mid \nu = 3\} \\
= &:: x : \textbf{int} \rightarrow y : \textbf{int} \rightarrow \{\nu : \textbf{bool} \mid \nu \Leftrightarrow (x = y)\} \\
+ &:: x : \textbf{int} \rightarrow y : \textbf{int} \rightarrow \{\nu : \textbf{int} \mid \nu = x + y\} \\
/ &:: x : \textbf{int} \rightarrow y : \{\nu : \textbf{int} \mid \nu \neq 0\} \rightarrow \{\nu : \textbf{int} \mid \nu = x/y\}
\end{aligned}$$

Fig. 3. Example Constants of the Language L1

**Definition 8 (Type Environment).** *The type environment is a mapping from program variables to their dependent types.*

The dependent types for the constants of the language — including binary operations — are predefined and listed in figure 3. The infix notation for binary operations is used for clarity. Some constants are defined regarding itself, e.g. the division operation is defined as $\nu = x/y$. These definitions are unproblematic since the refinement predicates are defined regarding the operational semantics used by the logic solver — as described in [6] — and not in the language L1.

The figure 4 shows the type inference rules of the language. The system has three different kinds of syntax directed judgments:

**Liquid Type Judgment** $\Gamma \vdash_{\mathbb{Q}} e : T$ These judgments are applied to expressions and state that an expression $e$ has the dependent type $T$ in the type environment $\Gamma$ when using the Qualifiers $\mathbb{Q}$.

**Well-Formedness Judgment** $\Gamma \vdash T$ The well-formedness judgments apply whenever a term of the form $\Gamma \vdash T$ is used inside of a liquid type judgment. The well-formedness judgments guarantee that the refinement predicate of the dependent type only refers to program variables in its scope — that are in the type environment Γ. Well-formedness judgments are explained in more detail in section 3.2.

**Subtype Judgment** $\Gamma \vdash T_1 <: T_2$ The subtyping judgment is a decidable and conservative subtyping rule stating that the dependent type $T_1$ is a subtype of $T_2$. The subtyping judgment rules are used together with the rule [T-SUB] to prove the subtyping relation between two dependent types. These rules are explained in more detail in section 3.2.

The next section uses the type inference rules to infer the dependent type for the *average throughput* function $\phi = s/t$ if applied with $s = 100$ and $t = 2$. The implementation of the $\phi$-function is defined in (4).

$$\textbf{let } \phi = \underbrace{(s \Rightarrow (t \Rightarrow s \;/\; t))}_{\textbf{int}\to\textbf{int}\to\textbf{int}} \textbf{ in } \underbrace{\phi\ 100\ 2}_{\textbf{int}} \tag{4}$$

## 3 Liquid Type Inference

The inference algorithm works similar to how mathematics exercises from preliminary school are solved. The steps needed to solve these exercises — and inferring the dependent types — are explained using the following example.

> If *Alice* doubles her age, she would still be ten years younger than *Bob*, who was born in 1952. How old are Alice and Bob? [7]

The first step is to create the *templates a* and *b* for the *unknown* age of Alice and Bob:

$$\begin{aligned} \text{Alice's age} &= a \\ \text{Bob's age} &= b \end{aligned} \tag{5}$$

The second step is to generate constraints capturing the relationship between the templates. For the given exercise, the relevant constraints for the templates $a$ and $b$ are:

$$\begin{aligned} 2a &= b - 10 \\ b &= 2016 - 1952 \end{aligned} \tag{6}$$

The last step is to find a solution for $a$ and $b$ that satisfies all the above constraints. One such solution is $a = 27$ and $b = 64$. Like the given exercise, liquid type inference is a constraint-based approach in which the constraints are implied by the program, e.g. what kind of values a variable might hold. The first step of the inference algorithm is to create the templates for all the unknown refinement predicates and is described in section 3.1. The second step — explained in section 3.2 — generates the constraints by inspecting the program. These constraints are solved in the final step described in section 3.3.

### 3.1 Template Generation

The first step in the preliminary mathematics exercise is to generate templates — variables — for the unknowns. A mapping for the unknown is needed to apply this analogy to liquid type inference. A dependent type is either a base type refined with a refinement predicate $p$ or a refined function type as defined in figure 2. To be able to know if it is the former or the latter, an algorithm is needed that can infer the ML-type $\tau$ of an expression. This algorithm is further denoted as *hm*.

**Remark 2.** *One such algorithm that infers the ML-type for a given expression and type checks the program according to the ML-typing rules — guaranteeing that the program is well-typed concerning the underlying type system — is the Hindley-Milner type inference algorithm. The algorithm is capable to automatically infer the ML-types without the need for any additional type annotations if the program is correctly typed but fails otherwise [8].*

A dependent type has the form $\{\nu : B \mid p\}$ if *hm* infers a base type $B$ for the expression and otherwise is a dependent function type. For both cases, the structure of the dependent type can immediately be determined. What is missing — the *unknown* — is the refinement predicate $p$ for dependent base types. According to the analogy, the first step of the inference algorithm is to generate templates — denoted as $F$ — for the unknowns. A template has the same form as a dependent type but uses a *liquid type variable* for the unknown refinement predicate.

**Definition 9 (Liquid Type Variable $\kappa$).** *A Liquid type variable $\kappa$ acts as a placeholder for the unknown refinement predicate of a dependent type.*

Liquid type variables are denoted as $\kappa$. Indexes are used to distinguish different liquid type variables. For example, $\kappa_x$ is the liquid type variable for the program variable $x$ and differs from $\kappa_2$ that is another liquid type variable — e.g. for the return type of a function.

There is only need to create a liquid type variable if a liquid type $\hat{T}$ is used in the inference rule, e.g. the rule [T-FUN] defines that a liquid type is needed for the argument $x$ and the return value. If not, then the dependent type can immediately be constructed from the types of the subexpressions. For example, the type of an application $e_1\ e_2$ can be composed by using the template for the called function $e_1$, taking the dependent type of the return value and substituting all occurrences of the parameter $x$ with the passed in argument $e_2$. This substitution is denoted as $[e_2/x]$.

A template for the $\phi$-function needs to be created whenever a liquid type $\hat{T}$ is used in an inference rule. The outermost expression is the let-expression for which the rule [T-LET] matches. The very first step is to infer the ML-type of the let-expression by calling *hm* which returns the type **int**. Therefore, the resulting template has to be a dependent base type with the structure $\{\nu : \textbf{int} | ?\}$. A liquid type variable is created for the refinement predicate of the body since it is still

**Liquid Type Judgment** $\boxed{\Gamma \vdash_{\mathbb{Q}} e : T}$

$$\frac{\text{true}}{\Gamma \vdash_{\mathbb{Q}} \mathrm{c} : ty(\mathrm{c})} \quad \text{T-CONST}$$

$$\frac{\Gamma(x) = \{\nu : B \mid p\}}{\Gamma \vdash_{\mathbb{Q}} x : \{\nu : B \mid \nu = x\}} \quad \text{T-VAR-BASE}$$

$$\frac{\Gamma(x) \text{ not a base type}}{\Gamma \vdash_{\mathbb{Q}} x : \Gamma(x)} \quad \text{T-VAR-FUN}$$

$$\frac{\Gamma; x : \hat{T}_x \vdash_{\mathbb{Q}} e : \hat{T} \qquad \Gamma \vdash x : \hat{T}_x \rightarrow \hat{T}}{\Gamma \vdash_{\mathbb{Q}} x \Rightarrow e : (x : \hat{T}_x \rightarrow \hat{T})} \quad \text{T-FUN}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : (x : T_x \rightarrow T) \qquad \Gamma \vdash_{\mathbb{Q}} e_2 : T_x}{\Gamma \vdash_{\mathbb{Q}} e_1\ e_2 : [e_2/x]T} \quad \text{T-APP}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : \mathbf{bool} \quad \Gamma; e_1 \vdash_{\mathbb{Q}} e_2 : \hat{T} \quad \Gamma; \neg e_1 \vdash_{\mathbb{Q}} e_3 : \hat{T} \quad \Gamma \vdash \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \hat{T}} \quad \text{T-IF}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : T_1 \quad \Gamma; x : T_1 \vdash_{\mathbb{Q}} e_2 : \hat{T} \quad \Gamma \vdash \hat{T}}{\Gamma \vdash_{\mathbb{Q}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \hat{T}} \quad \text{T-LET}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : T_1 \quad \Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2}{\Gamma \vdash_{\mathbb{Q}} e : T_2} \quad \text{T-SUB}$$

**Well-Formedness Judgment** $\boxed{\Gamma \vdash T}$

$$\frac{\Gamma; \nu : B \vdash p : \mathbf{bool}}{\Gamma \vdash \{\nu : B \mid p\}} \quad \text{WT-BASE}$$

$$\frac{\Gamma \vdash T_x \quad \Gamma; x : T_x \vdash T}{\Gamma \vdash x : t_x \rightarrow T} \quad \text{WT-FUN}$$

**Subtyping Judgment** $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{Valid([\![\Gamma]\!] \wedge [\![p_1]\!] \implies [\![p_2]\!])}{\Gamma \vdash \{\nu : B \mid p_1\} <: \{\nu : B \mid p_2\}} \quad \text{DEC-<:BASE}$$

$$\frac{\Gamma \vdash T_x' <: T_x \quad \Gamma; x : T_x' \vdash T <: T'}{\Gamma \vdash x : T_x \rightarrow T <: x : T_x' \rightarrow T'} \quad \text{DEC-<:FUN}$$

Fig. 4. Type Inference Rules for L1

unknown. This results in the template $\{\nu : \mathbf{int} \mid \kappa_{let}\}$ for the let-expression body.

Furthermore, the templates for the function abstraction of $\phi$ have to be created by applying the rule [T-FUN]. The rule is repeatedly applied for all arguments if the function is curried to reduce the number of steps. Invoking *hm* for the function abstraction returns the type $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$. Therefore, the template for the $\phi$-function has to be a function template since the inferred ML-type is a function type:

$$s : \{\nu : \mathbf{int} \mid \kappa_s\} \rightarrow (t : \{\nu : \mathbf{int} \mid \kappa_t\} \rightarrow \{\nu : \mathbf{int} \mid \kappa_{\phi ret}\}) \quad (7)$$

The Template uses liquid type variables for the unknown refinement predicates of the function parameters and body. This function template can be split into the three base type templates $F_s$, $F_t$, and $F_{\phi ret}$. The inference algorithm continues with the body of the throughput function. However, the body only contains expressions for which the resulting dependent type can immediately be constructed from the subexpressions. Therefore, no further templates are created. The created templates are summarized in figure 5.

$$\underbrace{\mathbf{let}\ \phi = (\overbrace{\overbrace{s}^{F_s} \Rightarrow \overbrace{t}^{F_t} \Rightarrow \overbrace{s\ /\ t}^{F_{\phi ret}}}^{F_\phi})\ \mathbf{in}\ \phi\ 100\ 2}_{F_{let}} \quad (8)$$

$$\begin{aligned} F_s &= \{\nu : \mathbf{int} \mid \kappa_s\} \\ F_t &= \{\nu : \mathbf{int} \mid \kappa_t\} \\ F_{\phi\ ret} &= \{\nu : \mathbf{int} \mid \kappa_{\phi\ ret}\} \\ F_{let} &= \{\nu : \mathbf{int} \mid \kappa_{let}\} \\ F_\phi &= s : F_s \rightarrow t : F_t \rightarrow F_{\phi ret} \end{aligned} \quad (9)$$

Fig. 5. Generated Templates for the Throughput-Function

## 3.2 Constraint Generation

The constraint generation step extracts constraints over the liquid type variables from the program. The extracted constraints create a relation between the liquid type variables. The idea is similar to the one of the preliminary mathematics exercise: Creating enough constraints so that the refinement predicates of the liquid type variables can be identified.

The algorithm distinguishes between well-formedness and subtyping constraints. The well-formedness constraint $\Gamma \vdash T$ — also named scope constraint — guarantees that a refinement predicate only refers to variables that are defined in the context it is used. The well-formedness constraint is semantically equal to the scoping rules of programming languages. An expression can only refer to variables in its scope. The scope of a dependent type $T$ is defined by the type environment $\Gamma$. The judgments for well-formedness constraints are specified in figure 4. A well-formedness judgment applies whenever a term of the form $\Gamma \vdash T$ is used inside of a liquid type judgment.

**Definition 10 (Well-Formedness Constraint $\Gamma \vdash T$).** *A well-formedness constraint requires that the refinement predicate of a dependent type be only over the program variables from the type environment and thus, is well-formed.*

The subtyping constraints describe the information flow in a program. The information flow for the if-expression of the *max* function from (10) is that the result may either hold the value of $x$ or $y$ dependent on whether the condition $x > y$ is true or not.

$$\textbf{let } max = \textbf{if } x > y \textbf{ then } x \textbf{ else } y \textbf{ in } max \ 10 \ 20 \tag{10}$$

In other words, the set of values of the *then* and *else* branches are a subset of the values of the entire if-expression — the values returned by the if-expression are the values of both branches. Therefore, the set of values for the if-expression is a superset of the values from the *then* and *else* branches ($Vals(T_{then}) \subseteq Vals(T_{if}), Vals(T_{else}) \subseteq Vals(T_{if})$). This is precisely a subtyping relation requiring that the types of the *then* and *else* branches be subtypes of the if-expressions type. The subtyping relation is denoted as $T_{sub} <: T_{sup}$. Liquid type inference creates a subtyping constraint to proof if a subtyping relation between two types holds. Therefore, a subtyping constraint is created whenever the rule [T-SUB] is used. The judgments for the subtyping constraints are shown in figure 4.

**Definition 11 (Subtyping Constraint $\Gamma \vdash T_1 <: T_2$).** *The subtyping constraint $\Gamma \vdash T_1 <: T_2$ states that the values of $T_1$ flow into $T_2$.*

**Remark 3.** *Inferring a precise type for the max function that considers the condition of the if-statement when inferring the refinement predicates for the then and else branches requires a path-sensitive algorithm as described in section 3.4.*

All needed well-formedness and subtyping constraints for the $\phi$-function application are summarized in figure 6. The rest of this section describes how these constraints are generated.

The rule [T-LET] requires that the type of the body $T$ be well-formed ($\Gamma \vdash T$) over the current type environment $\Gamma$. The body may refer to variables defined in the outer scope.

*Well-Formedness Constraints*:

$$\emptyset \vdash F_{let} \tag{11}$$
$$\emptyset \vdash F_s \tag{12}$$
$$s : F_s \vdash F_t \tag{13}$$
$$s : F_s; t : F_t \vdash F_{\phi ret} \tag{14}$$

*Subtyping Constraints*:

$$s : F_s; t : F_t \vdash F_s <: \{\nu : \textbf{int} \mid \text{true}\} \tag{15}$$
$$s : F_s; t : F_t \vdash F_t <: \{\nu : \textbf{int} \mid \nu \neq 0\} \tag{16}$$
$$s : F_s; t : F_t \vdash \{\nu : \textbf{int} \mid \nu = s/t\} <: F_{\phi ret} \tag{17}$$
$$\phi : F_\phi \vdash \{\nu : \textbf{int} \mid \nu = 100\} <: F_s \tag{18}$$
$$\phi : F_\phi \vdash \{\nu : \textbf{int} \mid \nu = 2\} <: F_t \tag{19}$$
$$\phi : F_\phi \vdash F_{\phi ret}[100/s][2/t] <: F_{let} \tag{20}$$

Fig. 6. Generated Well-Formedness and Subtyping Constraints

However, the variable defined by the let-expression itself is not part of the current type environment and can, therefore, not be referenced from the let body. This rule applied to the example results in one well-formedness constraint for $F_{let}$. The constraint requires that the refinement predicate only reference the special variable $\nu$ since the type environment is empty. The well-formedness constraint for the let expression is:

$$\emptyset \vdash F_{let} \tag{11}$$

The rule [T-FUN] requires that a well-formedness constraint be created for function abstractions. However, the well-formedness constraint for a function abstraction can be split into two well-formedness constraints — one for the parameter $x$ and another for the function body — as defined by the rule [DEC-<:FUN] shown in figure 4. The well-formedness constraint $\Gamma \vdash T_x$ requires the refinement predicate of the parameter $x$ only to refer to program variables from the outer scope. On the contrary, the well-formedness constraint for the function body $x : T_x \vdash T$ allows the refinement predicate to refer to the parameter $x$ and to the variables from the outer scope. The rule [T-FUN] applied to the $\phi$-function results in one well-formedness constraint for $s$, another for $t$ and a third for the function body. The constraint (12) for the argument $s$ requires that the refinement predicate for $F_s$ only contain constants or the special variable $\nu$ — as no other variables are in scope. The constraint (13) for $t$ allows the refinement predicate to refer to the variable $s$. The constraint (14) for the return type of the $\phi$-function restricts the referable variables by the refinement predicate to the variables $\nu$, $s$, and $t$.

$$\emptyset \vdash F_s \tag{12}$$
$$s : F_s \vdash F_t \tag{13}$$
$$s : F_s; t : F_t \vdash F_{\phi ret} \tag{14}$$

The data flow of the $\phi$-function is captured by subtyping constraints over the created templates. Let's start with the constraints for the division of $s$ by $t$. The arguments passed to the division function flow into its parameters requiring that the arguments be subtypes of the corresponding parameters. The type of the division operator is defined as a constant (see figure 3) and requires that the divisor not be equal to zero.

$$s : F_s; t : F_t \vdash F_s <: \{v : \textbf{int} \mid \text{true}\} \tag{15}$$
$$s : F_s; t : F_t \vdash F_t <: \{v : \textbf{int} \mid v \neq 0\} \tag{16}$$

The constraint (15) captures the fact that $s$ flows into the parameter $x$ of the division operation (figure 3) by requiring $F_s$ to be a subset of $\{v : \textbf{int} \mid \text{true}\}$ — an arbitrary **int** value.

The constraint (16) captures the flow of $t$ into $y$. The resulting type of the division $s/t$ is equal to the return type of the division operator. However, all occurrences of $x$ and $y$ need to be replaced — substituted — with the actual arguments $s$ and $t$. The refinement predicate would otherwise refer to variables that are not in scope. The resulting type of the division operation is $\{v : \textbf{int} \mid v = s/t\}$ in which all occurrences of $x$ are substituted with $s$ and the occurrences of $y$ with $t$.

Furthermore, the flow of the value returned by the division operation into the $\phi$-function's return value needs to be captured. Therefore, the type of the division operation needs to be a subtype of the function's return type $T_{div} <: F_{\phi ret}$. This data flow is captured by the subtyping constraint (17).

$$s : F_s; t : F_t \vdash \underbrace{\{v : \textbf{int} \mid v = s/t\}}_{T_{div}} <: F_{\phi ret} \tag{17}$$

Two subtyping constraints are created for the function application $\phi$ 100 2: constraint (18) captures the fact that 100 flows into $s$ and the constraint (19) the flow of 2 into $t$.

$$\phi : F_\phi \vdash \{v : \textbf{int} \mid v = 100\} <: F_s \tag{18}$$
$$\phi : F_\phi \vdash \{v : \textbf{int} \mid v = 2\} <: F_t \tag{19}$$

The type of the $\phi$-function application is equal to the return type of the $\phi$-function in which the parameters $s$ and $t$ are substituted with the actual arguments 100 and 2. The result of the application flows into the result of the let-expression. This flow is captured by the subtyping constraint (20) where $[100/s]$ denotes the substitution of the variable $s$ with the value 100.

$$\phi : F_\phi \vdash F_{\phi ret}[100/s][2/t] <: F_{let} \tag{20}$$

A summary of all generated well-formedness and subtyping constraints is given in figure 6. The next step is to find a solution for the liquid type variables $\kappa_s$, $\kappa_t$, $\kappa_{\phi ret}$, and $\kappa_{let}$ that satisfies all the constraints. This last step is explained in the next section.

## 3.3 Constraint Solving

The last step is to find a solution for $\kappa_s$, $\kappa_t$, $\kappa_{\phi ret}$, and $\kappa_{let}$ that satisfies all the generated constraints of figure 6. This step can also be explained by the analogy with the mathematics exercise. However, a new, more trivial example is used:

$$\begin{gathered} \mathbb{Q} = \{2, 3, 4, 5\} \\ 3 \leq x \leq 4 \end{gathered} \tag{21}$$

This exercise has only a single variable $x$, a single constraint, and the solution space is $\mathbb{Q}$. The solution of $x$ has to be a subset of $\mathbb{Q}$ and needs to fulfill the given constraint. A way to find the valid values for $x$ is to initialize the solution for $x$ with $\mathbb{Q}$ — all the possible values $x$ may hold. The next step is to verify if any constraint is not satisfied given the current solution by iterating over the constraints and testing if each constraint is satisfied. A value from the current solution of x is removed if it does not satisfy the current constraint. In the given example, the values 2 and 5 are withdrawn from the solution set as they impede the constraint from being satisfied. The remaining values form the solution since all constraints are satisfied. The final solution for $x$ is $\{3, 4\}$.

The same approach is applied to liquid type inference where variables map to liquid type variables, $\mathbb{Q}$ is the set of all possible qualifiers, and the constraints are the generated well-formedness and subtyping constraints. The current solution of each liquid type variable is stored in the liquid assignments map $A$. $A$ is a map from the liquid type variables to the set of qualifiers from $\mathbb{Q}^\star$.

**Definition 12 (Assignment Map $A(\kappa)$).** *The Assignment Map $A(\kappa)$ maps the liquid type variables to qualifiers from $\mathbb{Q}^\star$. It stores the current solution for each liquid type variable.*

The first step is to initialize the liquid assignment map using the set $\mathbb{Q}$ as the initial value for the liquid type variables. The set $\mathbb{Q}$ is used as the initial value because the solution of a liquid type variable has to be a subset thereof. The default initialized assignment map for the $\phi$-function application is shown in (22).

$$\begin{aligned}\kappa_s &\mapsto \{0 \le \nu, \nu \neq 0, \star \le \nu, \nu < \star\}\\ \kappa_t &\mapsto \{0 \le \nu, \nu \neq 0, \star \le \nu, \nu < \star\}\\ \kappa_{\phi ret} &\mapsto \{0 \le \nu, \nu \neq 0, \star \le \nu, \nu < \star\}\\ \kappa_{let} &\mapsto \{0 \le \nu, \nu \neq 0, \star \le \nu, \nu < \star\}\end{aligned} \tag{22}$$

The next step is to iterate over all constraints and to remove the qualifiers from the liquid assignment map that prevent a constraint from being satisfied. The iteration order is irrelevant, however, in this example, the well-formedness constraints are applied first to remove as many qualifiers as soon as possible. The Constraint (11) ensures that the refinement predicate of $\kappa_{let}$ only refers to $\nu$ but not to any other variable. The qualifiers $q$ from the current solution $A(\kappa_{let})$ are tested one by one if the constraint (11) is satisfied when $\kappa_{let} = q$. The constraint (11) is satisfied for the first qualifier $0 \le \nu$ as it only refers to the special variable $\nu$. The same holds for the second qualifier $\nu \neq 0$. The third qualifier contains the special $\star$-variable that can be instantiated with a program variable. However, the current constraint (11) prohibits the use of any program variable — the type environment is empty. Therefore, the qualifier prevents the constraint (11) from being satisfied and is thus removed. The same applies for the last qualifier also containing the $\star$-variable. The set of qualifiers for the second well-formedness constraint (12) over $\kappa_s$ are reduced equally. The resulting assignment map after testing the constraints (11) and (12) is:

$$\begin{aligned}\kappa_s &\mapsto \{0 \le \nu, \nu \neq 0\}\\ \kappa_t &\mapsto \{0 \le \nu, \nu \neq 0, \star \le \nu, \nu < \star\}\\ \kappa_{\phi ret} &\mapsto \{0 \le \nu, \nu \neq 0, \star \le \nu, \nu < \star\}\\ \kappa_{let} &\mapsto \{0 \le \nu, \nu \neq 0\}\end{aligned} \tag{23}$$

The constraint (13) requires that $t$ only reference $s$ or the special variable $\nu$. The embeddings of the first two qualifiers $0 \le \nu, \nu \neq 0$ hold. Not as before, the constraint (13) allows referencing the variable $s$. Therefore, the qualifier $\star \le \nu$ can be instantiated with $s$ resulting in the qualifier $s \le \nu$ for which the constraint holds. The same is true for the last qualifier.

The last well-formedness constraint (14) allows the qualifiers of $\kappa_{\phi ret}$ to refer to $s$ and $t$ as well. Therefore, the first two qualifiers, as well as the last two using the $\star$-variable, satisfy the constraint. The state of the assignment map after solving for all well-formedness constraints is shown in (24).

$$\begin{aligned}\kappa_s &\mapsto \{0 \le \nu, \nu \neq 0\}\\ \kappa_t &\mapsto \{0 \le \nu, \nu \neq 0, s \le \nu, \nu \le s\}\\ \kappa_{\phi ret} &\mapsto \{0 \le \nu, \nu \neq 0, s \le \nu, t \le \nu, \nu < s, \nu < t\}\\ \kappa_{let} &\mapsto \{0 \le \nu, \nu \neq 0\}\end{aligned} \tag{24}$$

The last step is to remove the qualifiers preventing the subtyping constraints from being satisfied. Here the question arises, how the subtyping constraint can be embedded into predicate logic — when does a subtyping constraint with a specific qualifier hold. Because the program is well-typed concerning the underlying type system, it can be assumed that the ML-type of the left- and right-hand side of a subtyping constraint are equal. Therefore, the subtyping constraint is reduced to test if the refinement predicates $p_{sub} <: p_{sup}$ satisfy the subtyping relation. This relation can be tested using the implication $p_{sub} \implies p_{sup}$. An implication holds if the first predicate is only true for values where the second predicate is too — the first predicate is never true for a value where the second is not. That exactly reflects the requirement of the subtyping constraint/relation; the subtype contains fewer values than the super-type.

However, the subtyping constraint also needs to consider the environment. Therefore, all variables of the current environment bound to base type templates (not function templates) are embedded into the logical expression. The subtyping constraint $x : \kappa_x \vdash \kappa_y <: \nu = 0$ is embedded as a conjunction of the current solution $A(\kappa_x)$ for $\kappa_x$ — and replacing all occurrences of $\nu$ with $x$ — and the tested qualifier. If the current solution set for $\kappa_x$ is $\{0 \le \nu, \nu \neq 0\}$ then the embedding for the subtyping constraint is the one shown in (25) in which $q$ is the qualifier to test if the current constraint is satisfied. Only qualifiers of the current solution $A(\kappa_y)$ are tested.

$$\underbrace{0 \le x \wedge x \neq 0}_{\Gamma \vdash x:\kappa_x[x/\nu]} \wedge \underbrace{q}_{q \in A(\kappa_y)} \implies \nu = 0 \tag{25}$$

**Remark 4.** *The rule [DEC-<:BASE] states that an implication is used to verify the subtyping relation of two refinement predicates. The notation* $[\![$ *and* $]\!]$ *in its premise express the embedding of the type environment and the refinement predicate into EUFA. EUFA stands for the decidable logic of equality, uninterpreted functions, and linear arithmetic [9].*

The subtyping constraints for the $\phi$-function application, their embedding into predicate logic and the result of testing the implication for every qualifier $q$ are summarized in table 1. Starting with the constraint (18) stating that the value 100 flows into $\kappa_s$. The embedding is $\nu = 100 \implies q$ since the value 100 flows into the value of $\kappa_s$. There is no embedding of the type environment as the type environment does not contain base type template mappings. The next step is to verify if the implication holds for every qualifier $q \in A(\kappa_s)$. The implication holds for both qualifiers since $0 \le 100$ and $100 \neq 0$.

The constraint (19) captures the flow of the value 2 into the parameter $t$. The first two implications hold. However, the implications for the qualifiers $0 \le s$ and $s \le \nu$ are not satisfied since the variable $s$ is undefined in the given environment.

The constraint (15) is the first with a non-empty type environment that needs to be embedded. The current solutions for $A(\kappa_t)$ and $A(\kappa_s)$ are embedded by replacing $\nu$ with $t$ or $s$. This constraint is trivially solved for every qualifier since the right-hand side is simply true.

| $\kappa$ | Subtyping-Constraint | | Embedding | $q \in A(\kappa)$ | SAT |
|---|---|---|---|---|---|
| $\kappa_s$ | $\phi : F_\phi \vdash \{v : \textbf{int} \mid v = 100\} <: F_s$ | (18) | $v = 100 \implies q$ | $0 \le v$ | ✓ |
| | | | | $v \neq 0$ | ✓ |
| $\kappa_t$ | $\phi : F_\phi \vdash \{v : \textbf{int} \mid v = 2\} <: F_t$ | (19) | $v = 2 \implies q$ | $0 \le v$ | ✓ |
| | | | | $v \neq 0$ | ✓ |
| | | | | $v \le s$ | ✗ |
| | | | | $s < v$ | ✗ |
| $\kappa_s$ | $s : F_s ; t : F_t \vdash F_s <: \{v : \textbf{int} \mid \text{true}\}$ | (15) | $\underbrace{0 \le s \wedge s \neq 0}_{s} \wedge \underbrace{0 \le t \wedge t \neq 0}_{t} \wedge q \implies \text{true}$ | $0 \le v$ | ✓ |
| | | | | $v \neq 0$ | ✓ |
| $\kappa_t$ | $s : F_s ; t : F_t \vdash F_t <: \{v : \textbf{int} \mid v \neq 0\}$ | (16) | $\underbrace{0 \le s \wedge s \neq 0}_{s} \wedge \underbrace{0 \le t \wedge t \neq 0}_{t} \wedge q \implies v \neq 0$ | $0 \le v$ | ✓ |
| | | | | $0 \neq 0$ | ✓ |
| $\kappa_{\phi ret}$ | $s : F_s ; t : F_t \vdash \{v : \textbf{int} \mid v = s/t\} <: F_{\phi ret}$ | (17) | $\underbrace{0 \le s \wedge s \neq 0}_{s} \wedge \underbrace{0 \le t \wedge t \neq 0}_{t} \wedge v = s/t \implies q$ | $0 \le v$ | ✓ |
| | | | | $v \neq 0$ | ✗ |
| | | | | $v \le s$ | ✓ |
| | | | | $v \le t$ | ✗ |
| | | | | $s < v$ | ✗ |
| | | | | $t < v$ | ✗ |
| $\kappa_{let}$ | $\phi : F_\phi \vdash F_{\phi ret}[100/s][2/t] <: F_{let}$ | (20) | $\underbrace{0 \le v \wedge v \le 100}_{F_{\phi ret}[100/s][2/t]} \implies q$ | $0 \le v$ | ✓ |
| | | | | $v \neq 0$ | ✗ |

Table 1. Results of Testing the Qualifiers against the Subtyping Constraints

The constraint (16) requires that the value $v$ be not equal to zero. The variables $s$ and $t$ are in scope, therefore are embedded into the implication. The implication is only tested for the qualifiers in the current solution of $A(\kappa_t)$. The implication for the qualifier $0 \le v$ on its own does not hold. However, the qualifier combined with $v \neq 0$ satisfies the constraint (16). Since the implication holds for both qualifiers together — the current solution — no qualifier needs to be removed.

**Remark 5.** *The qualifier $0 \le v$ would be eliminated according to the described algorithm that tests the qualifiers one by one and removes qualifiers preventing a constraint from being satisfied. However, the implementations of liquid types first test if the current solution satisfies the constraint and only remove qualifiers thereof if this is not the case. First testing the current solution results in preciser refinement predicates for the inferred liquid types.*

The constraint (17) captures the flow of the value returned by the $\phi$-function body into the return value of the $\phi$-function. The embedding of $s$ and $t$ guarantees that their values are larger than zero. Under this circumstances, the implication only holds for the qualifiers $\{0 \le v, v \le s\}$ as explained next: Dividing a positive number by another positive number results in a positive result that is less or equal to the quotient $s$ ($0 \le v, v \le s$). The result is equal to zero if the divisor $t$ is larger than the quotient $s$ since integer division is used ($v \neq 0$). The last three qualifiers make the implication unsatisfiable since the result might be larger than the divisor $t$ ($100/1 = 100$), the quotient $s$ can be equal to the result ($100/1 = 100$), and the divisor $t$ can be greater than the result ($1/100 = 0$).

The last constraint (20) captures the flow of the let body into the result of the entire let-expression. The difference to the constraint (17) is that the program variables $s$ and $t$ cannot be referenced. $F_{\phi ret}[100/s][2/t]$ denotes a substitution requiring that all occurrences of $s$ and $t$ in the current solution for $F_{\phi ret}$ be replaced with 100 and 2. The resulting implication only holds for $0 \le v$ since the left-hand side can be equal to zero.

The final state of the assignment map $A$ is:

$$\kappa_s \mapsto \{0 \le v, v \neq 0\} \tag{26}$$

$$\kappa_t \mapsto \{v \neq 0, v \le s\} \tag{27}$$

$$\kappa_{\phi ret} \mapsto \{0 \le v, v \le s\} \tag{28}$$

$$\kappa_{let} \mapsto \{0 \le v\} \tag{29}$$

The dependent types can be derived from the assignment map by connecting the qualifiers with a conjunction. For example, the dependent type for $F_s$ can be created by replacing $\kappa_s$ in the template by a conjunction of the current result in the assignment map $A(\kappa_s)$ resulting in the dependent type $\{\nu : \textbf{int} \mid 0 \leq \nu \wedge \nu \neq 0\}$. The inferred dependent types for the application of the $\phi$-throughput function with the values 100 and 2 are:

$$F_s = \{\nu : \textbf{int} \mid 0 \leq \nu \wedge \nu \neq 0\} \tag{30}$$
$$F_t = \{\nu : \textbf{int} \mid \nu \neq 0 \wedge \nu \leq s\} \tag{31}$$
$$F_{\phi\ ret} = \{\nu : \textbf{int} \mid 0 \leq \nu \wedge \nu \leq s\} \tag{32}$$
$$F_{let} = \{\nu : \textbf{int} \mid 0 \leq \nu\} \tag{33}$$

**Remark 6.** *Some of the inferred dependent types are more restrictive than they have to be that the program is considered safe. For example, the program is safe even when the refinement predicate for s is removed, allowing arbitrary **int** values. The refinement predicate of s — and all other liquid types — can be sliced away if it never occurs on the left-hand side of a subtyping constraint or if the right-hand side is simply true. It does not have to fulfill a specific subtyping constraint.*

If the $\phi$-function is instead applied with $\phi$ 100 0 then the verification fails. The subtyping constraint (19) capturing the data flow of the second argument into $t$ changes to the constraint (34) reflecting that 0 is passed instead of 2. The embedding of the changed constraint is shown in (35).

$$\phi : F_\phi \vdash \{\nu : \textbf{int} \mid \nu = 0\} <: F_t \tag{34}$$
$$\nu = 0 \implies q \tag{35}$$

This subtyping constraint is no longer satisfied by the qualifier $\nu \neq 0$ since $\nu = 0$ and is thus removed from $A(\kappa_t)$. Without this qualifier, the subtyping constraint (16) — requiring that $\nu \neq 0$ — does not hold for any qualifier that therefore, are all removed. At the end, the implication for the constraint (16) is reduced to the embedding (36). However, this implication is still not valid as the left-hand side gives no guarantees about $\nu$. Therefore, the program is ill-typed since no more qualifiers can be removed to make the implication to hold.

$$\underbrace{0 \leq s \wedge s \neq 0}_{s} \wedge \underbrace{0 \leq t}_{t} \implies \nu \neq 0 \tag{36}$$

## 3.4 Path Sensitivity

If the throughput function is extended to support time values equal to zero — in which case $\infty$ is returned — a guard is needed only to perform the division if $t \neq 0$ as shown in the following program:

```
let $\phi = s \Rightarrow t \Rightarrow$ if $t = 0$
    then $\infty$
    else $s\ /\ t$
    in $\phi$ 100 0
```

This program is still rejected by a non-path-sensitive type checker since the fact that $t$ is not equal to zero is not considered in the *else* branch. Therefore, path sensitivity is needed to capture the data flows precisely. Path sensitivity is added by extending the type environment to store the conditions encountered up to a particular point in the program as well. The *then* branch in this example can only be reached if the condition $t = 0$ is true.

The types of the *then* and *else* branches have to be subtypes of the type of the entire if-expression. Either the value of the *then* or *else* branches flows into the result of the if-expression. This flow is captured by the subtyping constraints (37) and (38).

$$s : F_s; t : F_t; \overbrace{(t = 0)}^{guard} \vdash F_{then} <: F_{if} \tag{37}$$
$$s : F_s; t : F_t; \underbrace{\neg(t = 0)}_{guard} \vdash F_{else} <: F_{if} \tag{38}$$

It is important to note that the subtyping constraints precisely captures when a branch/path is reached. The *then* branch is reached when the condition $t = 0$ is true. Therefore, the condition predicate is added as a guard to the type environment. The opposite applies to the *else* branch. For any predicate generated in the *then* or *else* branches, the guard predicate is added to the embedding of the constraint. For example, if the $\phi$-function is only applied with $t = 0$, then the guard $\neg(t = 0)$ added to the embedding (39) of the subtyping constraint (38) guarantees that the divisor is not equal to zero in this branch.

$$\underbrace{0 \leq s \wedge s \neq 0}_{s} \wedge \underbrace{t \leq 0 \wedge t = 0}_{t} \wedge \underbrace{\neg(t = 0)}_{guard} \wedge q \Rightarrow \nu \neq 0 \tag{39}$$

**Remark 7.** *It is assumed that $\mathbb{Q}$ is extended by the qualifier $\nu = 0$.*

The added qualifier guarantees that the left-hand side ends in a contradiction ($t = 0 \wedge \neg(t = 0)$), indicating that this is dead code and can safely be eliminated. The program is well-typed since the implication always holds.

Path sensitivity is also needed for the *max* function (10). The precision without path sensitivity degrades if the max function is applied twice, once with 20 and 0 ($x > y$), and the second time with 0 and 20 ($x < y$). The relevant, embedded, not path-sensitive subtyping constraints for the if-expression are:

$$\underbrace{0 \leq x}_{x} \wedge \underbrace{0 \leq y}_{y} \wedge \underbrace{v = x}_{then} \implies q \tag{40}$$

$$\underbrace{0 \leq x}_{x} \wedge \underbrace{0 \leq y}_{y} \wedge \underbrace{v = y}_{else} \implies q \tag{41}$$

Both constraints do not capture the relation between $x$ and $y$ resulting that the qualifier $y \leq v$ is removed by the constraint (40) and $x \leq v$ by the constraint (41), only leaving the qualifiers $\{0 \leq v, v \neq 0\}$. Including the condition from the if-expression as a guard in the embeddings (42) and (43) sets the variables $x$ and $y$ into a relation making the implication hold for the qualifiers $y \leq v$ and $x \leq v$ as well.

$$\underbrace{0 \leq x}_{x} \wedge \underbrace{0 \leq y \wedge}_{y} \wedge \underbrace{x > y}_{guard} \wedge \underbrace{v = x}_{then} \implies q \tag{42}$$

$$\underbrace{0 \leq x}_{x} \wedge \underbrace{0 \leq y}_{y} \wedge \underbrace{\neg(x > y)}_{guard} \wedge \underbrace{v = y}_{else} \implies q \tag{43}$$

This results in the preciser inferred dependent type shown in (44).

$$\{v : \textbf{int} \mid 0 \leq v \wedge x \leq v \wedge y \leq v\} \tag{44}$$

## 4 Conclusion

This paper has shown that dependent types are a valuable technique allowing type checkers to prove fine-granular invariants giving the programmer a better safety net. The additional precision of dependent types allows omitting many — no longer needed — runtime checks. However, dependent types have the displeasing property that type inference is undecidable and therefore, have to be manually annotated. This limitation can be overcome if type inference is reduced only to infer liquid types instead of dependent types.

This paper has further explained the liquid type inference algorithm. The algorithm uses Hindley-Milner to infer the ML-types and check if the program is well-typed in concern of the underlying type system — reducing dependent type inference to inferring the unknown refinement predicates. The algorithm creates templates containing liquid type variables as a placeholder for the unknown refinement predicates. In the next step, a system of constraints is extracted from the program that sets the liquid type variables into a relation. The final step is to solve the system of constraints. The program is ill-typed if any constraint is unsatisfiable and is well-typed otherwise. In the latter case, the qualifiers left in $A(\kappa)$ can be linked to a conjunction to create the dependent types.

The precision of the inferred refinement predicates can either be improved if the algorithm is path-sensitive or by adding more fine-granular qualifiers to $\mathbb{Q}$. Moreover, the programmer has always the option to manually annotate the dependent type if the inferred one is too imprecise.

## References

[1] M. Syfrig, “Dependent Types: Level Up Your Types,” *Program Analysis and Transformation*, 2016.

[2] G. Dowek, “The undecidability of typability in the Lambda-Pi-calculus,” in *Typed Lambda Calculi and Applications: International Conference on Typed Lambda Calculi and Applications TLCA ’93 March, 16–18, 1993, Utrech, The Netherlands Proceedings*, M. Bezem and J. F. Groote, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–145, ISBN: 978-3-540-47586-6. DOI: 10.1007/BFb0037103. [Online]. Available: http://dx.doi.org/10.1007/BFb0037103.

[3] B. C. Pierce, *Types and programming languages - the next generation*, 2003. [Online]. Available: http://www.cis.upenn.edu/~bcpierce/papers/tng-lics2003-slides.pdf (visited on 11/30/2016).

[4] C. Amrein, “Simply Typed Lambda Calculus with Parametric Polymorphism,” *Program Analysis and Transformation*, 2016.

[5] P. M. Rondon, M. Kawaguci, and R. Jhala, “Liquid Types,” *SIGPLAN Not.*, vol. 43, no. 6, pp. 159–169, Jun. 2008, ISSN: 0362-1340. DOI: 10.1145/1379022.1375602. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375602.

[6] K. Knowles and C. Flanagan, “Hybrid Type Checking,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 2, 6:1–6:34, Feb. 2010, ISSN: 0164-0925. DOI: 10.1145/1667048.1667051. [Online]. Available: http://doi.acm.org/10.1145/1667048.1667051.

[7] J. Ranjit. (Jul. 2008). Liquid Types, [Online]. Available: https://www.microsoft.com/en-us/research/video/liquid-types/ (visited on 10/28/2016).

[8] R. Milner, “A theory of type polymorphism in programming,” *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

[9] G. Nelson, “Techniques for program verification,” XEROX, Tech. Rep., 1981. [Online]. Available: https://people.eecs.berkeley.edu/~necula/Papers/nelson-thesis.pdf (visited on 11/17/2016).