Liquid Types

Manuel Eberl
<eberlm@in.tum.de>

April 29, 2013

1 Introduction

The purpose of type systems has always been to prevent certain errors, to guarantee, at compile time, certain run time behaviour of programmes. Type systems vary in their expressiveness, and consequently in the strength of the guarantees that they can provide. Simple, "dynamic" type systems, such as the one used by PHP, provide no guarantees whatsoever, whereas powerful type systems with dependent types, such as the one used by Idris, can express arbitrarily complex restrictions.

However, this expressiveness comes, of course, at the cost of undecidability. As a consequence of Rice's theorem, no non-trivial property of the behaviour of a programme is decidable, so type checking and type inference in expressive type systems is, in general, also undecidable. Idris solves this problem by demanding user-supplied proofs when the system cannot prove welltypedness automatically. Needless to say, this can get rather laborious.

To avoid this, most practical programming languages that provide a strong type system restrict themselves to less expressive types that can be checked and inferred automatically. The most commonly used functional programming languages – ML, OCaml and Haskell – use higher order static type systems, sometimes with some extensions such as polymorphism, existential types and type classes, but often, the user would like to specify stronger restrictions: for instance, one could imagine a division function that requires the divisor to be nonzero, or an array access function that requires the index to be in range. In most programming languages, these values trigger (unwanted) runtime exceptions. Of course, it would be much better to have a type system that *guarantees* that such errors cannot occur at runtime, by rejecting programmes that e.g. use division with a potentially zero divisor. Naturally, while some overapproximation is necessary, one would also like such a type system to be "clever" enough to not reject too many correct programmes either.

Liquid Types is such a type system. It provides types that are powerful enough to express restrictions like these, but still weak enough for type checking and type inference to work well automatically in most practical cases.

2 Definitions

Refinement types are central to the idea of *liquid types*. A refinement type is a certain subtype of a "regular" type, such as *Int* or *List*, with an additional constraint on the value, such as $\{\nu : \text{Int} \mid \nu \neq 0\}$ or $\{\nu : \alpha \text{ List} \mid \text{len}(\nu) = 5\}$[1]. Refinement types can be used to encode preconditions and postconditions of functions; one example would be the *replicate* function, which takes some value v of type a and a non-negative integer n and returns a list of length n containing only the value v, e.g. *replicate* $2\ 3 = [2, 2, 2]$. In a type system with refinement types, a possible type of this function that captures the property that the length of the list *replicate* produces always has exactly as many elements as requested, could be:

$$\text{replicate} :: a \to n : \{\nu : \text{Int} \mid \nu \geq 0\} \to \{\nu : \text{List}\, a \mid \text{len}(\nu) = n\}$$

With this, the typechecker will reject any implementation of *replicate* that does not satisfy this property. Also, the compiler may be able to infer that access to elements of that list with an index of $< n$ are safe, which guarantees safety and makes compile-time optimisation of the accesses possible.

The idea of liquid types is now to restrict the form of these contraints to a conjunctions of a small set of atomic *qualifiers* from a decidable fragment of the underlying logic, which allows efficient type checking and inference. For instance, the original paper [RKJ08] uses the following set of qualifiers, or "qualifier templates" as an example:

$$\mathbb{Q} = \{0 \leq \nu, * \leq \nu, \nu < *, \nu < \text{len}(*)\}$$

[1] ν simply stands for "the value we are talking about at the moment", i.e. the value is described by the type

The $*$ symbol here is a placeholder for any variable in the context of the expression in question, whereas ν is a placeholder that refers to the value that the type refers to. Consider, for instance, a function with integer parameters x and y. The set of valid qualifiers is obtained from the templates by instantiating $*$ with all appropriate variables in the context – in this case x and y:[2]

$$\mathbb{Q}_\Gamma = \{0 \leq \nu, x \leq \nu, \nu < x, y \leq \nu, \nu < y\}$$

Any conjunction of qualifiers from this set is a valid liquid type. For instance, $0 \leq \nu \wedge \nu < y \wedge \nu < x$ would be valid, but $\nu < 0$ or $0 \leq \nu \vee \nu < y$ would be invalid. The concrete set of qualifiers $\mathbb{Q}$ varies depending on the nature of the guarantees that the liquid types are supposed to provide, but it should contain only qualifiers that the theorem prover can handle. Also, the more qualifiers are allowed, the more computation time type checking/inference will require. The implementation of the authors of the original paper uses constraints of the form $\nu \bowtie X$, where $\bowtie$ is any of the relations $\leq$, $<$, $=$, $>$, $\geq$, $\neq$, and X is 0, $*$, or $\mathrm{len}(*)$. More structural properties about algebraic datatypes, such as $\nu \neq$ *None* can also be added.

3 Type inference

Liquid type inference works in three steps:

Hindley-Milner type inference: this is the "standard" type inference algorithm, which can be seen as inferring an "overapproximation" of the desired liquid types where all liquid type constraints are set to *True*. Consequently, the following steps should refine these to more precise types. In order to do that, every "basic" type[3] assigned a liquid type variable κ that contains constraints that are yet to be determined. For instance, Int $\to$ Int $\to$ Int becomes $x : \{\nu : \text{Int} \mid \kappa_1\} \to y : \{\nu : \text{Int} \mid \kappa_2\} \to \{\nu : \text{Int} \mid \kappa_3\}$. This is referred to as the *liquid type template* and may be abbreviated as $x : \kappa_1 \to y : \kappa_2 \to \kappa_3$ when the basic types are irrelevant or clear from the context.

Liquid constraint generation: in this step, a set of rules[4] is applied to the programme in order to find constraints for the liquid type variables. The rules fall into three categories:

- **Liquid type checking rules:** syntax-directed rules that produce subtyping and wellformedness constraints
- **Subtyping rules:** required to solve the subtyping obligations produced by the liquid type checking rules
- **Wellformedness rules:** a constraint is well-formed if it only contains valid qualifiers, instantiated with variables from the context. In particular, the constraints must, of course, not contain any variables that do not occur in the context.

To keep things simple, we will ignore the wellformedness rules and implicitly assume that we always have well-formed liquid types.

Constraint solving The system of constraints is solved iteratively using an SMT solver. Initially, all liquid type variables are set to the strongest value possible, i.e. the conjunction of all valid qualifiers. The solving algorithm then looks for any unsatisfied constraints and weakens the liquid type variables involved so that the constraint is fulfilled, until no unsatisfied constraints remain.[5]

This approach is called *iterative weakening*, as we start out with the strongest restrictions possible on every basic type in our programme and weaken the assignment just enough for the constraints to be fulfilled. The result is an assignment that assigns every liquid type variable the most restrictive liquid type possible in order to provide the maximum amount of information about the values.

Let us now look at the constraints in question. Since we ignored the wellformedness constraints, the only "interesting" kind of constraints is the subtyping constraints: a constraint of the form $\Gamma \vdash \kappa_1 <: \kappa_2$ requires that κ_2 is implied by the context Γ and the liquid type variable κ_1. For instance, if $\Gamma = x : \{\nu : \text{Int} \mid \nu \geq 0\}$ and $\kappa_1 = x < \nu$ and $\kappa_2 = 0 < \nu$ the statement $\Gamma \vdash \kappa_1 <: \kappa_2$ is true, since $x \geq 0$ and $\nu > x$ implies $\nu > 0$. Since we restricted ourselves to a finite number of qualifiers and a decidable logic, we can resolve an unsatisfied subtyping constraint $\Gamma \vdash \kappa_1 <: \kappa_2$ by looking at all qualifiers in κ_2 one by one and, using a decision procedure for our logic, removing those that are not implied by Γ and κ_1. This approach makes sure that we only weaken the κ_i as much as we absolutely have to.

[2] The $\nu < \mathrm{len}(*)$ qualifier was not instantiated here, because it requires $*$ to be an array and x and y were assumed to be integers

[3] "Basic" meaning "non-composed". For instance, the composed type Int $\to$ Int contains two basic types, namely two Int.

[4] for the exact rules, refer to the original paper by Rondon et al. [RKJ08]

[5] Or there are no more constrants that can be weakened, in which case the algorithm outputs an error message. For example, if we have a *replicate* function with the type as above as a rigid type constraint and call it with the second parameter -1, the algorithm would have to weaken our rigid constraint, which it must not do.

4 Example

As an example, take the following programme:

$$max\ (a :: \text{Int})\ (b :: \text{Int}) = \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

This can be seen as the definition of the constant *max* as:

$$\lambda a :: \text{Int}.\ \lambda b :: \text{Int}.\ \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

Hindley-Milner tells us that the type of the *max* function is:

$$\text{Int} \to \text{Int} \to \text{Int}$$

Therefore, the liquid type template is:

$$a : \{\nu : \text{Int} \mid \kappa_1\} \to b : \{\nu : \text{Int} \mid \kappa_2\} \to \{\nu : \text{Int} \mid \kappa_3\}$$

The next step is constraint generation. We have to use the liquid typing rules on the expression[6]

$$\textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a$$

in the context

$$\Gamma = [a : \{\nu : \text{Int} \mid \kappa_1\};\ b : \{\nu : \text{Int} \mid \kappa_2\}]$$

The context tells us what variables exist and what types they have. In our case, we know they are integers and that they fulfil the (still unknown) liquid predicate κ_1 resp. κ_2. Since we have an **if** expression, we need to apply the corresponding rule for typing **if** expressions:

$$\frac{\Gamma \vdash_{\mathbb{Q}} a < b : \text{Bool} \qquad \Gamma; a < b \vdash_{\mathbb{Q}} b : \kappa_3 \qquad \Gamma; b \leq a \vdash_{\mathbb{Q}} a : \kappa_3}{\Gamma \vdash_{\mathbb{Q}} \textbf{if}\ a < b\ \textbf{then}\ b\ \textbf{else}\ a\ :\ \kappa_3}$$

The premise about $a < b$ can be ignored, since we already know it is well-typed from Hindley-Milner; the interesting part is the other two premises: They say that in order to derive some liquid type κ_3 for the **if** expression, we have to show that both the **then** branch and the **else** branch return a value that satisfies this liquid predicate κ_3, but we can use the fact that the **if** condition is fulfilled (resp. not fulfilled) as an additional assumption when doing this, so this assumption is added to the context. This feature is known as *path sensitivity*.

Let us now look at the first condition, the one for the **then** branch: intuitively, we know that b has type κ_2, but we are required to show that it has type κ_3. The only way to show this is to show that κ_2 is a subtype of κ_3, i.e. κ_3 is a more general property than κ_2. This is done using the subtyping rule, which demands that, in the given context, κ_2 implies κ_3:

$$\frac{\dfrac{}{\Gamma; a < b \vdash_{\mathbb{Q}} b : \{\nu : \text{Int} \mid \nu = b\}} \qquad \dfrac{\Gamma; a < b \vDash \nu = b \Rightarrow \kappa_3}{\Gamma; a < b \vdash \{\nu : \text{Int} \mid \nu = b\} <: \{\nu : \text{Int} \mid \kappa_3\}}}{\Gamma; a < b \vdash_{\mathbb{Q}} b : \{\nu : \text{Int} \mid \kappa_3\}}$$

The proof of the condition of the **else** branch is analogous:

$$\frac{\dfrac{}{\Gamma; b \leq a \vdash_{\mathbb{Q}} a : \{\nu : \text{Int} \mid \nu = a\}} \qquad \dfrac{\Gamma; b \leq a \vDash \nu = a \Rightarrow \kappa_3}{\Gamma; b \leq a \vdash \{\nu : \text{Int} \mid \nu = a\} <: \{\nu : \text{Int} \mid \kappa_3\}}}{\Gamma; b \leq a \vdash_{\mathbb{Q}} a : \{\nu : \text{Int} \mid \kappa_3\}}$$

If we look at the very top of these trees, we see that the following two conditions remain:

$$\Gamma; a < b \vDash \nu = b \Rightarrow \kappa_3 \quad \text{and} \quad \Gamma; b \leq a \vDash \nu = a \Rightarrow \kappa_3$$

These are now the conditions we have to work with. Ordinarily, the constraint solving algorithm would initialise all the κ_i with all qualifiers in $\mathbb{Q}_\Gamma$, and since the function is not called in our programme, there are no constraints for κ_1 and κ_2, they would remain this way. For demonstration purposes, let us therefore assume we have some other function in our programme that actually calls the *max* function with integer values about which we know nothing (i.e. $\{\nu : \text{Int} \mid \mathit{True}\}$). In that case, we will have the additional constraints:

$$\mathit{True} \Rightarrow \kappa_1 \quad \text{and} \quad \mathit{True} \Rightarrow \kappa_2$$

[6] To be more precise, we first have to apply the liquid typing rule for λ abstraction to the expression, but these do nothing except "stripping away" the λ and adding the bound variables to the context, so we skipped this step.

These constraints are, of course, unsatisfied for the initial assignment of all qualifiers, which is why the algorithm has to weaken them down to the empty set of qualifiers, i.e. *True*.

We then have to worry about our original constraints, which, with the current assignment of $[\kappa_1 \mapsto \text{True}, \kappa_2 \mapsto \text{True}]$ are basically equivalent to:

$$(a < b \Rightarrow \kappa_3) \quad \text{and} \quad (b \leq a \Rightarrow \kappa_3)$$

The theorem prover iterates over all the allowed qualifiers for κ_3 and finds that the only ones that fulfil the constraints are $a \leq \nu$ and $b \leq \nu$, leading to the following type for max:

$$a : \{\nu : \text{Int} \mid \text{True}\} \rightarrow b : \{\nu : \text{Int} \mid \text{True}\} \rightarrow \{\nu : \text{Int} \mid a \leq \nu \wedge b \leq \nu\}$$

5 In practice

There are a number of real world implementations of Liquid Types for different languages. The authors of the original paper [RKJ08] gave an implementation for OCaml. Another very sophisticated implementation is *LiquidHaskell* [Jha], which supports a great number of additional features, such as liquid types for algebraic datatypes and user-defined liquid predicates; an online demo can be found at `http://goto.ucsd.edu/~rjhala/liquid/haskell/demo/`.

References

[Jha] Ranjit Jhala, *Liquid Haskell*, `http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/about/`.

[RKJ08] Patrick M. Rondon, Ming W. Kawaguchi, and Ranjit Jhala, *Liquid types*.

[Ron12] Patrick M. Rondon, *Liquid types*, Ph.D. thesis, University of California, 2012.