Towards a most general type

October 24, 2013

Intro

Consider a function

```
f :: Ord a => a -> a
f x = assert (x > 0) x
```

The type liquidHaskell will infer for this function, depends on its call sides. So, if `f` is called with positive arguments, we will get

```
f :: Ord a => {x:a | v > 0} -> {v:a | v = x}
```

If `f` is called with any argument, we will get

```
f :: Ord a => x:a -> {v:a | v = x}
```

and an error on the definition of `f`.

Even worse, if `f` is not called at all, we will get

```
f :: Ord a => {v:a | false} -> {v:a | false}
```

Even though this behaviour is not modular, as we shall see it is reasonable, it can lead to more precise types and most importantly, it allows for efficient type inference.

Liquid Type Inference

Liquid type inference has two important features, namely logical qualifiers and refinement variables.

Refinement Variables

Liquid Types use the common rules (see FORMALISM) for type-checking and subtyping but whenever the system should **guess** a type they use haskell's type as template to create a liquid type with variables as refinements.

As an example, in the rule

$$\frac{\Gamma, x{:}\tau_x \vdash e{:}\tau \qquad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x.e{:}(x : \tau_x \to \tau)}$$

Say that haskell's (unrefined) type of the variable x is a type variable a, then the liquid type will be

$$\{v{:}a \mid k_x\}$$

and the well formedness constraint

$$\Gamma \vdash \{v{:}a \mid k_x\}$$

will state that k_x can **depend** on any variable that exists on Γ.

Qualifiers

But what can k_x be solved to? It should be solved to a logical predicate. Since we can not search every possible predicate, it will be a conjunction of *logical qualifiers*. Logical qualifiers, are given as input to the solver and are of the following form

$$\begin{array}{rrcl} \textbf{\textit{Qualifiers}} & q & ::= & x(\overline{x}) := p \\ \textbf{\textit{Predicates}} & p & ::= & e[> \mid \geq \mid < \mid \leq \mid == \mid !=]e \;\mid\; p[\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow]p \;\mid\; \neg p \\ \textbf{\textit{Expressions}} & e & ::= & x \;\mid\; c \;\mid\; e[+ \mid -]e \end{array}$$

As an example, the logical qualifier

$$q(x) = v > x$$

compares the qualified value v with the variable x.

Well-formedness Constraints

We create and split well-formedness rules, as usual.

Using the basic well-formedness constraints `WFC` for all refinement variables and the set of qualifiers `Q` we decide all the possible candidates of the variables as follows:

$$cand(k) = \{q(\overline{x}) \mid \Gamma \vdash_B \{v{:}\tau \mid k\} \in \texttt{WFC} \wedge \Gamma, v : \tau \vdash q(\overline{x}){:}bool\}$$

We note that in $q(\overline{x})$ the variables $\overline{x}$ instantiate the qualifier's variables with concrete ones that appear in the well-formedness constraints. Also, typechecking $\Gamma \vdash e{:}\tau$ is the usual unrefined typechecking rule.

$$\begin{array}{rrcl} \textbf{\textit{Well-Formedness Constraints}} & \texttt{WFC} & ::= & \emptyset \;\mid\; \Gamma \vdash_B \tau, \texttt{WFC} \\ \textbf{\textit{Qualifiers}} & \texttt{Q} & ::= & \emptyset \;\mid\; q(\overline{x}), \texttt{Q} \end{array}$$

Sub-Typing Constraints

We create and split well-formedness rules, as usual (see FORMALISM).

As an example, in the rule

$$\frac{\Gamma \vdash e_1{:}(x : \tau_{x1} \rightarrow \tau) \qquad \Gamma \vdash e_2{:}\tau_{x2} \qquad \Gamma \vdash \tau_{x2} \preceq \tau_{x1}}{\Gamma \vdash e_1\ e_2{:}\tau}$$

We state that the type of the argument τ_{x2} should be a substype of the argument of the function τ_{x1}. Through substyping rules, constraints split down to implication checking of the form

$$\Gamma \vdash r_1 \Rightarrow r_2$$

Where r is a refinement, ie., a list of either a refinement variable or a predicate.

Implication Constraints	SUB	::=	$\emptyset \mid \Gamma \vdash r \Rightarrow r, \texttt{SUB}$
Refinement	r	::=	$\emptyset \mid q, r$
Basic Refinement	q	::=	$k \mid p$

Solving Implications

We use the below fixpoint algorithm to solve refinement variables:

```
st := {k, cand(k)};

update:
 forall G |- r1 => r2 in SUB
  let p1 := {p | p in r1} `union` {st(k) | k in r1}
  forall k in r2
   st(k) := st(k) `intersection` p1
 if st is updated go to update
```

Finally, we use a solver to prove that all constraints are satisfied.

In this algorithm the refinement variables start from the stronger solution and at each iteration they can get weaker so at to satisfy the constraints. So, the final solution we get is the stronger with respect to the constraints.

Weakest Preconditions

The fact liquid inference returns the stronger solution, means that we get the strongest preconditions to functions, which can be undesirable. As an example, in the function

```
qualif Bot(v:a)      = 0 =  1
qualif EqZero(v:a)   = v =  0
qualif GtZero(v:a)   = v >  0
qualif GEZero(v:a)   = v >= 0
qualif Cmp(v:a, x:a) = v >= x

f :: Ord a => a -> a
f x = assert (x > 0) x
```

we shall give a type

```
f :: Ord a => {v:a | k_x} -> {v:a | k_r}
```

`k_x` will be initialized with all the appropriate qualifiers, `k_x := [0=1, v = 0 , v > 0, v >=0]` and since it does not appear in the right hand side of any constraint, this will be its final solution, so `k_x := false`.

To address this issue, we named the refinement variables that appear in function arguments as *negative* (and the rest positive), and tried to provide the weakest solution for negative variables.

One approach

Our first approach was to treat negative variables in a way dual to positives: Initialize them to the weakest solution, ie $k^- ::= []$ and update them every time they appear in the left hand side.

This approach quickly failed as the left and the right sides of the implication are not symmetric:

The constraint $\Gamma \vdash r_1 \Rightarrow r_2$ is interpreted as

$$\bigwedge \{r[v/x] | \{x : v : a | r\} \in \Gamma\} \wedge r_1 \Rightarrow r_2$$

So in order to satisfy this constraint we can either weaken the (positive) variables in the right-hand side or strengthen the variables the variables in the left-hand side or the environment.

In the above example, verification proceeds as

```
assert :: {v : Bool | v <=> true} -> a -> a
(>) :: Ord a => x:a -> y:a -> {v : Bool | v <=> x > y}
```

Thus, from this code we get a constraint

```
{x:a | k_x}
|-
(v <=> x > 0) => (v <=> true)
```

This constraint indicates, that in order to give a meaningful type to `f` it does not suffice to update the negative variables in the implication, but we have to update the variables in the environment.

One can say that we could update only the variables that refine variables appear in the implication, like `x`, but this is not always the case.

Consider the code

```
g :: Ord a => a -> a -> a
g x y = if x > 0 then x else assert (y > 0) y
```

This code leads to the implication:

```
{x:a | k_x}
{y:a | k_y}
not (x > 0)
|-
(v <=> y > 0) => (v <=> true)
```

One `y` appears in the implication, but we can get numerous safe types that restrict either `k_x` or `k_y`. Some examples are, `k_x := []` and `k_y := [v > 0]` and `k_x := [v > 0]` and `k_y := []`

The problem is that these two types are **not comparable**. Even though two solutions of one variable can be easily compared, (as they form a lattice,) this is not the case for more than one variables.

A natural ordering would be to use lexicographic ordering, but this fails as `k_x := []` and `k_y := [0=1]` will always be a solution, but since it actually prevents the function to be called, it is stronger than `k_x := [v > 0]` and `k_y := []`.

Since we cannot order solutions we cannot choose one solution over another we would have to search for all possible solutions and combine them. Otherwise, the behaviour of the solver would not be predictable.

In the example above, there are **four** basic solutions, ie, none of which implies the other:
`k_x := []` and `k_y := [v > 0]`,
`k_x := [v = 0]` and `k_y := []`,
`k_x := []` and `k_y := [v = x, v >= 0]`, and
`k_x := [x >= 0]` and `k_y := [v >= x]`

We can combine all these solutions, using $\vee$, to compose the solution with the most general precondition:

```
g :: Ord a
  => x:a
  -> {y:a | v > 0 || x = 0 || (v = x && v >= 0) || (x
     >= 0 && v >= x)}
  -> a
```

Searching the lattices

According to the above discussion, in the search of the most general precondition, we have to find all the possible solutions. To do so, we need to search all the possible combinations of all the lattices of the refinement variables. Moreover, if there exist n refinement variables we can order two solutions as

$$(q_1, \ldots, q_n) \sqsubseteq (p_1, \ldots, p_n) \text{ iff } q_1 \sqsubseteq p_1 \wedge \cdots \wedge q_n \sqsubseteq p_n$$

The above ordering makes more solutions incomparable, which prevents us from finding a clever way to search the solution space. For example if an assignment is solution, we know that we need to search stronger solutions. Dually is some assignment is not a solution, we know that all weaker assignments can not be solutions. But since most assignments are not comparable, these heuristics can not be used to make our search more efficient.

Obviously, searching all the solution space is too slow. To typecheck `g` we need 1.6 seconds. Worse, time increases exponentially, with respect to the number of refinement variables and the number of qualifiers.

As an example, consider the following code

```
g :: x:Int -> y:[a] -> w:[a] -> z:Int -> Int
g x y w z = assert (z > 0) z
```

There are 8 negative variables in the above code, as each list type has three refinements: one to describe list properties, like list's length, one to describe properties of its elements, ie., a list of positive numbers, and one to describe recursive relations of lists, ie., an increasing list. So, we need to combine solutions from 8 lattices.

Even worse, the more variables exist in the environment, the more qualifiers can be used to solve the refinement variable. So, even though only 9 qualifiers can describe x, giving a lattice of 22 independent solutions, there are ?? qualifiers that can describe z, giving a lattice of independent solutions. All in all we should search 8 lattices of sizes ??? just to export a trivial precondition.

Weakest Precondition and Most General Solution

An observation is that most general preconditions do not lead to most general types, as they restrict the predicates of the result. As an example, consider the following function

```
maxInt      :: Int -> Int -> Int
maxInt x y = if x > y then x else y
```

This function enforces no preconditions, so, we can set all negative variables to true and get a type:

```
maxInt :: Int -> Int -> Int
```

Another type is one that assuming that both arguments are positive returns a positive result:

```
maxInt :: {v:Int|v>0} -> {v:Int|v>0} -> {v:Int|v>0}
```

In the second alternative, the result type is stronger. Thus, these types can not be compared according to subtyping relation.

LiquidHaskell can choose the second type, if `maxInt` is called with two positive values.

Comparing these two types, the first one has greater domain and is modular, ie., the inferred type doesn't not depend on functions call sides. On the other hand the second one can be more informative, as using it you can prove more properties. On the same time, inferring the first type is much more expensive. Thus, choosing the second type is not worst than choosing the first.

FORMALISM

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & e & ::= & x \mid c \mid \lambda x.e \mid e\ e \\
\textbf{\textit{Refined Types}} & \tau & ::= & \{v : b \mid r\} \mid x : \tau \to \tau \\
\textbf{\textit{Environment}} & \Gamma & ::= & \emptyset \mid x{:}\tau, \Gamma
\end{array}
$$

$$\Gamma \vdash e{:}\tau$$

$$\frac{(x, \{v : b \mid r\}) \in \Gamma}{\Gamma \vdash x{:}\{v : b \mid v = x\}} \qquad \frac{(x, y : \tau_y \to \tau) \in \Gamma}{\Gamma \vdash x{:}(y{:}\tau_y \to \tau)}$$

$$\frac{}{\Gamma \vdash c{:}ty(c)} \qquad \frac{\Gamma \vdash e_1{:}(x : \tau_{x1} \to \tau) \quad \Gamma \vdash e_2{:}\tau_{x2} \quad \Gamma \vdash \tau_{x2} \preceq \tau_{x1}}{\Gamma \vdash e_1\ e_2{:}\tau}$$

$$\frac{\Gamma, x{:}\tau_x \vdash e{:}\tau \quad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x.e{:}(x : \tau_x \to \tau)}$$

$$\Gamma \vdash \tau$$

$$\frac{\Gamma \vdash_B \{v : b \mid r\}}{\Gamma \vdash \{v : b \mid r\}} \qquad \frac{\Gamma \vdash \tau_x \quad \Gamma, x{:}\tau_x \vdash \tau}{\Gamma \vdash x : \tau_x \to \tau}$$

$$\Gamma \vdash \tau \preceq \tau$$

$$\frac{\Gamma, v : b \vdash r \Rightarrow r'}{\Gamma \vdash \{v : b \mid r\} \preceq \{v : b \mid r'\}} \qquad \frac{\Gamma \vdash \tau'_x \preceq \tau_x \quad \Gamma, x{:}\tau'_x \vdash \tau \preceq \tau'}{\Gamma \vdash x : \tau_x \to \tau \preceq x : \tau'_x \to \tau'}$$