1 Refinement Types

This section is primarily based on Principles of Type Refinement, Noam Zeilberger, OPLSS 2016

The concept of *refinement types* is quite general. So general, in fact, that it is not immediately obvious that the various publications claiming to present refinment type systems are founded upon a common framework. Nonetheless, Noam Zeilberger recently published a tutorial presenting such a framework. Before shifting to a more historical perspective, we will examine Zeilberger's framework so that we understand the presented systems as *refinement* type systems rather than ad-hoc ones.

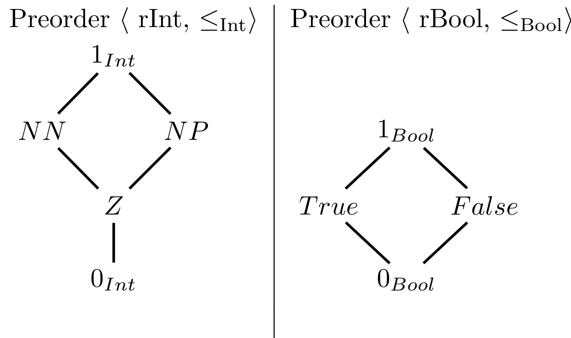
Definition 1.1. *Refinement type system:* An extra layer of typing on an existing type system.

Remark 1.2. Type refinement systems are *conservative*: they preserve the properties of the underlying type system. The framework to be presented assumes the existing type system does *not* include subtyping.

Definition 1.3 (Type syntax)**.**
T, S, U (types) ::= Int | Bool | $T \to T$
ρ, π, σ (refinements) ::= rInt | rBool | $\rho \to \rho$

Remark 1.4. A refinement type system decomposes each base type of the underlying system into a preordered set of base refinement types. In our example system, these sets (rInt and rBool) happen to be lattices.

Definition 1.5 (Base refinements)**.**



Remark 1.6 (Type semantics)**.** Recall that a type T can be interpreted as a set of closed values; e.g., $[\![Bool]\!] = \{true, false\}$ where $true$ and $false$ are values in a term language. For distinct S and T the underlying system should have

$[\![T]\!] \cap [\![S]\!] = \emptyset$.

Remark 1.7 (Form of refinement semantics)**.** For refinement types ρ, $[\![\rho]\!]$ is a pair (A, B) where

1. $\exists S.A = [\![S]\!]$
2. $B \subseteq A$

We can think of types as non-overlapping boxes and refinements as areas inside these boxes which "know which boxes they are in".

Definition 1.8 (Refinement relation $\rho \sqsubset \tau$)**.**

$$\frac{\rho \in rInt}{\rho \sqsubset Int} \qquad \frac{\rho \in rInt}{\rho \sqsubset Int} \qquad \frac{\rho \sqsubset T \qquad \pi \sqsubset S}{\rho \to \pi \sqsubset T \to S}$$

Remark 1.9. Induction on the derivation of $\rho \sqsubset T$ shows that $\sqsubset$ is *functional*: $\rho \sqsubset T \wedge \rho \sqsubset S \Rightarrow T = S$.

Remark 1.10. At base types, $\sqsubset$ inuitively corresponds to the set inclusion relation. But isn't that what subtyping is for? Noting that R-FUN has a *co-variant* left-hand premise, we see that our intuition isn't quite correct. If $\rho \sqsubset S$ and $\pi \sqsubset T$ then $\rho \to \pi$ identifies the set of lambdas in $[\![S \to T]\!]$ which map *snd* $[\![\rho]\!]$ into *snd* $[\![\pi]\!]$. $NN \to 1_{\text{Int}} \sqsubset Int \to Int$ simply because $NN \sqsubset Int$ and $1_{\text{Int}} \sqsubset Int$. $[\![NN \to 1_{\text{Int}}]\!] = ([\![Int \to Int]\!], [\![Int \to Int]\!])$.

$\rho \sqsubset T$ means "the purpose of ρ is to indentify a subset of $[\![T]\!]$".

Remark 1.11. Our payoff is a subtyping relation between refinement types. Recall that our underlying type system has no subtyping relation and that for distinct S and T, $[\![S]\!] \cap [\![T]\!] = \emptyset$. Subtyping judgments between refinements of distinct underlying types are therefore useless. For this reason we only consider subtyping judgments of the form $\rho <:_T \pi$, in which both $\rho \sqsubset T$ and $\pi \sqsubset T$. This inspires the mantra *refinement comes before subtyping*.

Definition 1.12 (Subtyping relation $\rho <:_T \pi$:)**.**

$$\frac{\rho \leq_{Int} \pi}{\rho <:_{Int} \pi} \qquad \frac{\rho \leq_{Bool} \pi}{\rho <:_{Bool} \pi} \qquad \frac{\pi_1 <:_T \rho_1 \qquad \rho_2 <:_S \pi_2}{\rho_1 \to \rho_2 <:_{T \to S} \pi_1 \to \pi_2}$$

Remark 1.13 (Semantic subtyping)**.** If $\rho <:_T \pi$ then there exist sets X and Y such that $[\![\rho]\!] = ([\![T]\!], X)$, $[\![\pi]\!] = ([\![T]\!], Y)$, and $X \subseteq Y$.

Definition 1.14 (Term syntax)**.** t ::= $x \mid \lambda x : T.t \mid t\ t$

Definition 1.15 (Typing relation $\Gamma \vdash t : T$)**.** Standard rules elided.

Definition 1.16 (Refinement contexts Δ)**.** $\Delta ::= \cdot \mid \Delta, x : \rho$

Definition 1.17 (Context refinement $\Delta \sqsubset \Gamma$)**.** $\Delta \sqsubset \Gamma$ when $\Delta = x_1 : \rho_1, ..., x_n : \rho_n$, $\Gamma = x_1 : T_1, ..., x_n : T_n$, and $\rho_1 \sqsubset T_1, ..., \rho_n \sqsubset T_n$.

Definition 1.18 (Refinement typing relation $\Delta \vdash t : \rho$)**.** Standard App & Var.

$$\frac{\Delta, x : \pi \vdash t : \rho \qquad \pi \sqsubset T}{\Delta \vdash \lambda x : T.\ t : \pi \to \rho}\ \text{ABS}$$

$$\frac{\Delta \sqsubset \Gamma \qquad \Gamma \vdash t : T \qquad \rho, \pi \sqsubset T \qquad \Pi \vdash t : \rho \qquad \rho <:_T \pi}{\Delta \vdash t : \pi}\ \text{SUB}$$

Definition 1.19 (Principal types)**.** Let $\Gamma \vdash t : T$, $\Pi \sqsubset \Gamma$, and $\rho \sqsubset T$. ρ is a *principal type* of t under Δ when for all π with $\Delta \vdash t : \pi$ we have $\rho <:_T \pi$.

Example 1.20 (Lack of principal types)**.** Does $(\lambda x : int.\ x)$ have a principal type under the empty refinement context?

$$\emptyset \vdash (\lambda x : int.\ x) : 1_{\text{Int}} \to 1_{\text{Int}}$$

and

$$\emptyset \vdash (\lambda x : int.\ x) : 0_{\text{Int}} \to 0_{\text{Int}}$$

but any common subtype $\rho \to \pi$ of $1_{\text{Int}} \to 1_{\text{Int}}$ and $0_{\text{Int}} \to 0_{\text{Int}}$ must have $1_{\text{Int}} <:_{int} \rho$ and $\pi <:_{int} 0_{\text{Int}}$. The only such type is $1_{\text{Int}} \to 0_{\text{Int}}$, but

$$\emptyset \not\vdash (\lambda x : int.\ x) : 1_{\text{Int}} \to 0_{\text{Int}}$$

$(\lambda x : int.\ x)$ therefore has no principal type.

2 Refinement Types for ML

Freeman and Pfenning, ACM SIGPLAN 1991.

Remark 2.1. ML, Haskell, and other languages with pattern matching have a weakness that programmers are constantly running into: it's often reasonable to make an assumption about a case scrutinee, the precision of which exceeds that of the type system. Here is a simple example:

```
type List = Nil | Cons of Int * List
*empty space*
fun lastcons (last as Cons(hd, Nil)) = last
  | lastcons (Cons(hd,tl)) = lastcons t1
```

The lastcons function returns the last cons of a non-empty list. But when supplied with nil, it will raise an inexhaustive match exception. Further consider how we might use an application of lastcons.

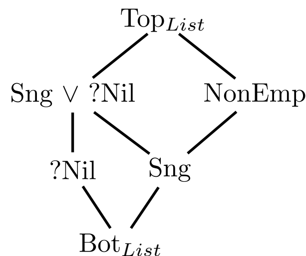
```
case lastcons y of
  Cons(x,nil) => print x
```

A type checker for ML or Haskell would produce an "inexhaustive match" warning for such a case expression, but examining the code we see that lastcons *only* returns values matching the pattern cons(x,nil). How can we solve this problem?

Remark 2.2. Freeman and Pfenning's solution is to layer a refinement system on ML's underlying type system, in which each discriminated union data type is decomposed into a preorder (a finite lattice in this case) of base refinements. The structure of each lattice is user defined using an extended type definition language.

Remark 2.3. *fill in existing list type definition*

```
type List = Nil | Cons of Int * List
rectype Sng = Cons (Int, Nil)
rectype NonEmp = Cons (Int,List)
```



$?Nil$, a list refinement denoting the singleton set containing only the value Nil, is necessary due to the definition of Sng. We'll soon see why it is necessary to add the list refinement $Sng \vee ?Nil$.

Remark 2.4. With our new lattice of list refinements, we can provide our list constructors with more precise types. We start with naive refinements, inferred from the declaration "type list = Nil | Cons of Int * List", which add no extra precision:

$$Nil : 1_{List}$$

$$Cons : 1_{Int \to List \to List}$$

The declaration “rectype single = cons (Int,Nil)“ gives

$$cons : Int \;*\; ?Nil \to Sng$$

These types are then combined into intersections.

$$cons : (1_{Int \to List \to List}) \wedge (Int \;*\; ?Nil \to Sng)$$

Remark 2.5. The inclusion of single ∨ ?nil in our refinement lattice allows case expressions to utilize the extra precision of refinement types. A case expression in which the scrutinee is a list is desugared into a call to a function of the following type:

CASE_list : $\forall\, 'a.\ \forall\, 'ra_1 :: 'a.\ \forall\, 'ra_2 :: 'a.$

scrutinee	Nil case	Cons case	result type	
(Sng ∨ ?Nil)	→ (Unit → $'ra_1$)	→ (Int * ?Nil → $'ra_2$)	→ ($'ra_1 \vee 'ra_2$)	∧
Sng	→ (Unit → $'ra_1$)	→ (Int * ?Nil → $'ra_2$)	→ $'ra_2$	∧
?Nil	→ (Unit → $'ra_1$)	→ (Int * 1_{list} → $'ra_2$)	→ $'ra_1$	∧
1_{list}	→ (Unit → $'ra_1$)	→ (Int * 1_{list} → $'ra_2$)	→ ($'ra_1 \vee 'ra_2$)	

We instantiate the refinement variable $'ra_1$ with the type of the nil case body, and $'ra_2$ with the type of the cons case body. If the scrutinee has type 1_{list} then we don't know whether it was constructed with nil or cons, so the type of the case expression (in the fourth column) must be the union of the types of the bodies of the nil and cons cases.

Remark 2.6 (Type inference)**.** As the complexity of rectype definitions grows, the complexity of the types of programs written under those definitions explodes. For this reason Freeman and Pfenning developed a type inference system based on abstract interpretation.

3 Dependent ML

Xi, Journal of Functional Programming, 2004.

Remark 3.1. The ability to distinguish empty lists and singletons is well and good, but it would be more useful if we could have a distinct List refinement for every possible length. Refining base types via terms of a computationally tractable constraint language, called an *index language* gives us this power. It enables the following precise types for the append operation:

$$append : \{\text{n,m : int}\}\ \text{List(m)} \to \text{List(n)} \to \text{List(n+m)}$$

Remark 3.2. Dependent ML is parameterized over an index language. The most practical one, which I will be using in my examples, consists of linear arithmetic expressions over integers, constrained by inequalities. Reasoning

about such constraints amounts to linear integer programming which, while NP-complete, can be solved quite efficiently in practice. The "types" of index terms will be called sorts. Base sorts will be denoted with the metavariable s:

s (base sorts) := bool | int

The terms of our index language, denoted with the metavariable I, are algebraic, consisting of index variables and applications of constants. It contains 0-arity constants for int and bool literals, as well as standard arithmetic operators such as $+, -, *, =$, and $>$, and propositional logic operators $\neg, \wedge$, and $\vee$. A constant's sort is a multi-argument function of the form $(\vec{s}) \rightarrow s$.

Remark 3.3. Dependent ML provides a *restricted form of dependent types*, in that types depend on index terms rather than program terms. We use $\vec{P}$ to denote a *constraint set* (a set of index terms of sort bool) and ϕ to denote a context of index variables. Subtyping and typing judgments then have the following forms.

$\phi; \vec{P} \vdash \rho <: \pi$ and
$\phi; \vec{P}; \Gamma \vdash t : \rho$

Constraints, types, and terms depend on index variables.

$\vec{P} \overset{constrains}{\longrightarrow} \phi \overset{refined\ by}{\longleftarrow} \Gamma, t, \rho$

Example 3.4.

```
type List (int) = nil(0) | {n : int} cons(n+1) of Int * List(n)
```

Here is the Dependent ML version of our List type. It is refined using an index term of sort int; List(n) is the List refinement corresponding to all List values of length n. The nil constructor produces a value of type List(0), while the cons constructor is wrapped in an index abstraction; given an index n of sort int, a value of type Int, and a value of type List(n), cons produces a value of type List(n+1).

Example 3.5. We now rewrite the type ascriptions of the lastcons function into the style of Dependent ML.

```
fun lastcons {m : int | m > 0} (l : List(n)) : List(1) =
  case l of
  | cons(hd, nil) = l
  | cons(hd, tl as Cons(_,_)) = lastcons tl
```

When type checking the base case, where l matches $cons(hd, nil)$, we add the index variable $n : int$ into our index context ϕ. We're assuming l has length

m, but $cons(hd, nil)$ has length $n + 1$, so we add constraint $m = n + 1$ to $\vec{P}$. We expect the second argument of *cons* to have length n, but because it has length 0 we add $n = 0$ to $\vec{P}$. The body l of this case clearly has refinement type $List(n)$, but result has been ascribed the refinement type $List(1)$; we therefore require the following subtyping judgment to hold:

$$m : int, n : int; m > 0, m = n + 1, n = 0 \vdash List(n) <: List(1)$$

It holds due to the following inference rule:

$$\frac{\phi; \vec{P} \vDash I = J}{\phi; \vec{P} \vdash \delta(I) <:_\delta \delta(J)} \ \text{ST-SUB-BASE}$$

The premise says that constraint set $\vec{P}$ and index terms I and J are well-sorted under index context ϕ, and also that the constraint solver can prove $\vec{P}$ entails $I = J$. The conclusion says that for any base type δ, δ-indexed-with-I is a subtype of δ-indexed-with-J.

Remark 3.6. This rule expands the refinement types framework in some interesting ways. Consider *List*'s preordered set of refinements $\langle rList, \leq_{List} \rangle$. $rList$ is the set of index terms which are well-sorted under ϕ, and so $rList$ is not static throughout the type-checking process; it is instead a function of the sorting context ϕ! Also, $\leq_{List}$ varies with respect to ϕ and $\vec{P}$. For example, under $m : int, n : int; \emptyset$ we have $List(m) \not\leq_{List} List(n)$, while under $m : int, n : int; m = n$ we have $List(m) \leq_{List} List(n)$. In Dependent ML, then, we should not associate each base type with a preordered set, but instead a family of preordered sets indexed by ϕ and $\vec{P}$.

Remark 3.7. If you thought it strange that we only committed to imposing preorders on each base type rather than partial orders or lattices, Dependent ML shows that such generality pays off. Under any ϕ and $\vec{P}$ the refinements of a base type are ordered by an *equivalence*, which is not even a partial order.

Remark 3.8. I'm now going to attempt to understand Dependent ML through the lens of Noam's framework. This was not discussed in the paper, but as I mentioned earlier, I think it's important to understand all of papers discussed as instances of the same formal framework. Getting back to our example, the type of the lastcons function is $\Pi(x : int).(n > 0) \supset (List(n) \rightarrow List(1))$. It is composed of refinement type constructors of the form $\Pi(x : int).\rho$, $P \supset \rho$, and $\rho \rightarrow \rho$.

The refinement type constructor $\Pi(x : int).\rho$ is the type of index abstractions. The refinement type constructor $P \supset \rho$ is the type of *guarded terms*, which intuitively are terms of type ρ that can only be accessed when the constraint set $\vec{P}$ entails P. In Dependent ML's internal language, $P \supset \rho$ has the introduction form $\supset^+ (v)$ and the elimination form $\supset^- (t)$. Their non-algorithmic typing rules follow:

$$\frac{\phi; \vec{P}, P; \Gamma \vdash v : \rho}{\phi; \vec{P}; \Gamma \vdash \supset^{+} (v) : P \supset \rho} \text{ TY-}\supset\text{-INTRO} \qquad \frac{\phi; \vec{P}; \Gamma \vdash t : P \supset \rho \qquad \phi; \vec{P} \vDash P}{\phi; \vec{P}; \Gamma \vdash \supset^{-} (t) : \rho} \text{ TY-}\supset\text{-ELIM}$$

Which underlying type T does $P \supset \rho$ refine? If $\rho \sqsubset S$ then $P \supset \rho$ should refine a type T such that $[\![T]\!] = \{\supset^{+} (v) \mid v \in [\![S]\!]\}$. We can call this type $* \supset S$, read *stub implies S*. DML never defines such a type, and in fact the type system underlying DML's internal language is not defined. However, such a system is implicit both in the internal language's term syntax and in a subtyping relation referred to as the *static subtyping relation*.

Remark 3.9. Isn't DML supposed to refine ML, which has no stub implication types? Yes, and in fact DML defines a external language which *does* refine ML if we're willing to relax our definition of refinement. The refinement relation of the external language leverages the intuition that in the above remark $[\![S]\!]$ and $[\![T]\!]$ are isomorphic as sets. But we should only consider them interchangeable if they are isomorphic with respect to our refinement system; i.e., the following two judgments must be derivable:

$$\phi; \vec{P}; x : \rho \vdash \supset^{+} (x) : P \supset \rho$$

and

$$\phi; \vec{P}; x : P \supset \rho \vdash \supset^{-} (x) : \rho$$

which will be the case whenever we have

$$\phi; \vec{P} \vDash P$$

This suggests that we must modify our refinement types framework once again. Now a refinement ρ will be interpreted as a pair (A, B) where:

1. $\exists S.[\![S]\!] \cong A$
2. $B \subseteq A$

A refinement judgment for our external language then takes the form

$$\phi; \vec{P} \vdash \rho \sqsubset S$$

and means that under sorting context ϕ and constraint set $\vec{P}$, ρ identifies a subset of some set that is *isomorphic* to S with respect to our refinement type system. The inference rules for this relation would include the following:

$$\frac{\phi; \vec{P} \vdash \rho \sqsubset S \qquad \phi; \vec{P} \vDash P}{\phi; \vec{P} \vdash P \supset \rho \sqsubset S} \text{ R-IMP}$$

Xi defines a *dynamic subtyping* over this refinement relation. It is not sound to use values of two isomorphic sets interchangeably; $\supset +(v)$ can't be used where v

is expected. For this reason the dynamic subtyping relation produces "instructions", in the form of an evaluation context E, for mapping terms of refinement type $P \supset \rho$ into terms of refinement type ρ. Its subtyping judgments therefore take the form $\phi; \vec{P} \vdash E : \rho <:_S \pi$. It contains a rule (dy-sub-Π-l) which subsumes the functionality of the following simplified rule:

$$\frac{\phi; \vec{P} \vdash E : \rho <:_S \pi \qquad \phi; \vec{P} \vDash P}{\phi; \vec{P} \vdash \supset^- (E) : P \supset \rho <:_S \pi}$$

4 Liquid Haskell

Vazou et al. Refinement types for Haskell. ACM SIGPLAN Notices. Vol. 49. No. 9. ACM, 2014.

Remark 4.1. *Liquid types* refers to a method for inferring refinement types. It involves a constraint solving algorithm that is tangential to the topic of this lecture. Instead, I will examine how the work of Jhala and his students fits into the refinement types framework.

Remark 4.2. Instead of expressing constraints in an index language, liquid type systems use the term language itself. Here is an example of a base refinement in Liquid Haskell:

$$\{v : Int \mid v > 0\}$$

Here the constraint $v > 0$ is a program term rather than an index term. We can interpret this type as the set of integers n such that $[n/v](v > 0) \Downarrow true$. A subtyping judgment between base refinements then requires the supertype's constraint to normalize to true whenever the subtype's constraint does.

$$\frac{\forall \gamma \in [\![\Gamma]\!]. \gamma e_1 \Downarrow true \implies \gamma e_2 \Downarrow true}{\Gamma \vdash \{v : B \mid e_1\} <: \{v : B \mid e_2\}}$$

The idea of refining types with arbitrary program terms originated in Cormac Flanagan's Hybrid Typechecking, which dealt with the undecidability of the above subtyping rule's premise by falling back on contract enforcement techniques. Liquid Haskell conservatively approximates such implication checks with a logic of equality, uninterpred functions, and linear integer arithmetic.

Remark 4.3. An uninterpreted function is a multi-argument function which allows a single law, called *congruence*, for reasoning about its applications. If F is an uninterpreted function, then congruence tells us:

$$\forall t_1, \ldots, t_n, t'_1, \ldots, t'_n. \bigwedge_i (t_i = t'_i) \implies F(t_1, \ldots, t_n) = F(t'_1, \ldots, t'_n)$$

In a pure language like Haskell, we can encode function applications as applications of uninterpreted functions. In ML this technique is not applicable because any function may produce side effects.

Example 4.4. Reworking our running example into the style of liquid types gives:

```
--[[ (comment annotation)
@measure
len : List -> Int
len nil = 0
len cons(hd, tail) = (length tail) + 1
]]

fun lastcons (l : { v : List | len(v) > 0 }) : {v : List | len(v) = 1} =
  case l of
  | cons(hd, nil) = l
  | cons(hd, tail) = lastcons tail
```

Remark 4.5. We can think of len as a program function. However, adding the @measure annotation to len provides our constraint solver access to extra axioms when it is encoded into an uninterpreted function. These axioms are contained in the types of the List constructors:

```
nil  : {v : List | len(v) = 0}
cons : hd : Int -> tail : List -> {v : List | len(v) = len(tail) + 1}
```

Remark 4.6. Walk through the above example.

Remark 4.7. The typing context Γ contains not just variable bindings $x : \rho$, but also boolean term constraints t which encode *path sensitivity*, i.e. information about which branch was taken in a case expression. The subtyping judgment

$$l : \{v : \mathit{List} \mid \mathit{len}(v) > 0\}, (\mathit{len}(l) = 1) \vdash \{v : \mathit{List} \mid v = l\} <: \{v : \mathit{List} \mid \mathit{len}(v) = 1\}$$

must hold for the first case body to type check. It is translated into the following formula.

$$\underbrace{(\mathit{len}(l) > 0) \wedge (\mathit{len}(l) = 1)}_{\Gamma} \wedge \underbrace{(v = l)}_{lhs} \implies \underbrace{(\mathit{len}(v) = 1)}_{rhs}$$

Remark 4.8. Because Liquid Haskell has no extra type syntax structure for managing an index language, it has a very simple refinement relation.

$$\frac{\Gamma, v : B \vdash t : \mathit{Bool}^{\Downarrow}}{\Gamma \vdash \{v : B \mid t\} \sqsubset B}$$

$\mathit{Bool}^{\Downarrow}$ is the type of all boolean terms which normalize.

Remark 4.9. If B is a base type, B has a Γ-indexed family of refinements preorders, $\langle rB, \leq_B \rangle_\Gamma$