Liquid Meta(l)

MICHAEL BORKOWSKI
(summarizing joint work with RANJIT JHALA and NIKI VAZOU)

December 27, 2019

1 Our language λ_1

We work with a simply typed lambda calculus which is augmented by refinement types, dependent function types, and existential types. Our language is based on λ in [Jhala] and λ^U in [Vazou].

We start with the syntax of term-level expressions in our language:

$$\begin{array}{llll}
\textsf{Values} & v ::= & \texttt{true}, \texttt{false} & \textit{boolean constants} \\
 & \mid & 0, 1, 2, \ldots & \textit{integer constants} \\
 & \mid & x & \textit{variables} \\
 & \mid & \lambda x.e & \textit{abstractions} \\
 & \mid & e_1 \wedge e_2,\ e_1 \vee e_2,\ \neg e_1 & \textit{built-in primitives} \\
 & \mid & e_1 \leq e_2,\ e_1 = e_2 & \textit{built-in primitives}
\end{array}$$

$$\begin{array}{llll}
\textsf{Expressions} & e ::= & v & \textit{values} \\
 & \mid & e_1\ e_2 & \textit{applications} \\
 & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \textit{let expressions} \\
 & \mid & e_1 : t & \textit{annotations}
\end{array}$$

Next, we give the syntax of the types and binding environments used in our language:

$$\begin{array}{llll}
\textsf{Base types} & b ::= & \textsf{Bool} & \textit{booleans} \\
 & \mid & \textsf{Int} & \textit{integers}
\end{array}$$

$$\begin{array}{llll}
\textsf{Types} & t ::= & b & \textit{base} \\
 & \mid & b\{r\} & \textit{refinement} \\
 & \mid & x{:}t_x \rightarrow t & \textit{dependent function} \\
 & \mid & \exists\, x{:}t_x.\, t & \textit{existential}
\end{array}$$

$$\textsf{Refinements} \quad r ::= \{v : p\}$$

$$\begin{array}{llll} \textsf{Environments} & \Gamma ::= & \varnothing & \textit{empty} \\ & \mid & \Gamma, x{:}t & \textit{bind variable} \end{array}$$

Next, we give the syntax of the Boolean predicates and constraints involved in refinements and subtyping judgments:

$$\begin{array}{llll} \textsf{Predicates} & p ::= & \{e \mid \exists\,\Gamma.\,\Gamma \vdash e : \textsf{Bool}\} & \textit{expressions of type } \textsf{Bool} \end{array}$$

$$\begin{array}{llll} \textsf{Constraints} & c ::= & p & \textit{predicates} \\ & \mid & c_1 \wedge c_2 & \textit{conjunction} \\ & \mid & \forall\, x : b.\, p \Rightarrow c & \textit{implication} \end{array}$$

Our definition of predicates above departs from the languages of [Jhala] by allowing predicates to be arbitrary expressions from the main language (which are Boolean typed under the appropriate binding environment). In [Jhala] however, predicates are quantifier-free first-order formulae over a vocabulary of integers and a limited number of relations. We initially took this approach, but were unable to fully define the denotational semantics for this type of language. In particular, when we define closing substitutions we need to define the substitution of a type $\theta(t)$ as the type resulting from t after performing substitutions for all variables bound to expresssions in $\theta = (x_1 \mapsto e_1, \ldots, x_n \mapsto e_n)$. Substituting arbitrary expressions into t requires substituting arbitrary expressions into predicates, and it isn't clear how to do this for non-values like $((\lambda x.x)\,3)$ without taking predicates to be all Boolean-typed program expressions.

Returning to our λ_1, we next define the operational semantics of the language. We treat the reduction rules (small step semantics) of the various built-in primitives as external to our language, and we denote by $\delta(c, v)$ a function specifying them. The reductions are defined in a curried manner, so for instance we have that $c\ v_1\ v_2 \hookrightarrow^* \delta(\delta(c, v_1), v_2)$. Currying gives us unary relations like $m\leq$ which is a partially evaluated version of the $\leq$ relation.

$$\begin{array}{rlrl} \delta(\wedge, \texttt{true}) &:= \lambda x.\, x & \delta(\leq, m) &:= m\leq \\ \delta(\wedge, \texttt{false}) &:= \lambda x.\, \texttt{false} & \delta(m\leq, n) &:= \texttt{bval}(m \leq n) \\ \delta(\vee, \texttt{true}) &:= \lambda x.\, \texttt{true} & \delta(=, m) &:= m= \\ \delta(\vee, \texttt{false}) &:= \lambda x.\, x & \delta(m=, n) &:= \texttt{bval}(m = n) \\ \delta(\neg, \texttt{true}) &:= \texttt{false} & & \\ \delta(\neg, \texttt{false}) &:= \texttt{true} & & \end{array}$$

Now we give the reduction rules for the small-step semantics. In what follows, e and its variants refer to an arbitrary expression, v refers to a value, x to a variable, and c refers to a built-in primitive.

$$\frac{}{c\ v \hookrightarrow \delta(c,v)}\ \textsc{E-Prim} \qquad \frac{e \hookrightarrow e'}{e\ e_1 \hookrightarrow e'\ e_1}\ \textsc{E-App1}$$

$$\frac{e \hookrightarrow e'}{v\ e \hookrightarrow v\ e'}\ \textsc{E-App2} \qquad \frac{}{(\lambda x.\, e)\ v \hookrightarrow e[v/x]}\ \textsc{E-AppAbs}$$

$$\frac{e_x \hookrightarrow e_x'}{\texttt{let}\, x = e_x\, \texttt{in}\, e \hookrightarrow \texttt{let}\, x = e_x'\, \texttt{in}\, e}\ \textsc{E-Let} \qquad \frac{}{\texttt{let}\, x = v\, \texttt{in}\, e \hookrightarrow e[v/x]}\ \textsc{E-LetV}$$

$$\frac{e \hookrightarrow e'}{e : t \hookrightarrow e' : t}\ \textsc{E-Ann} \qquad \frac{}{v : t \hookrightarrow v}\ \textsc{E-AnnV}$$

Next, we define the typing rules of our λ_1. As with the reduction rules, we take the type of our built-in primitives to be external to our language. We denote by $ty(c)$ the function that specifies them:

$$\begin{aligned}
ty(\wedge) = ty(\vee) &:= \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
ty(\neg) &:= \mathsf{Bool} \to \mathsf{Bool} \\
ty(\leq) = ty(=) &:= \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
ty(m{\leq}) = ty(m{=}) &:= \mathsf{Int} \to \mathsf{Bool} \\
ty(\texttt{true}) = ty(\texttt{false}) &:= \mathsf{Bool} \\
ty(n) &:= \mathsf{Int}
\end{aligned}$$

The type judgments in the language λ_1 will be denoted $\vdash$ with a colon between term and type. For clarity, we distinguish between this and other judgments by using $\vdash$ with a subscript in most other settings. For instance, the judgement $\Gamma \vdash_w t$ says that type t is well-formed in environment Γ:

$$\frac{\Gamma, v{:}b \vdash e : \mathsf{Bool}}{\Gamma \vdash_w b\{v : e\}}\ \textsc{WF-Base} \qquad \frac{\Gamma \vdash_w t_x \qquad \Gamma, x{:}t_x \vdash_w t}{\Gamma \vdash_w x{:}t_x \to t}\ \textsc{WF-Func}$$

$$\frac{\Gamma \vdash_w t_x \qquad \Gamma, x{:}t_x \vdash_w t}{\Gamma \vdash_w \exists\, x{:}t_x.\, t}\ \textsc{WF-Exis}$$

Now we give the rules for the typing judgements:

$$\frac{ty(c) = t}{\Gamma \vdash c : t}\ \textsc{T-Prim} \qquad \frac{x{:}t \in \Gamma}{\Gamma \vdash x : t}\ \textsc{T-Var} \qquad \frac{\Gamma, x{:}t_x \vdash e : t \qquad \Gamma \vdash_w t_x}{\Gamma \vdash \lambda x.\, e \ :\ x{:}t_x \to t}\ \textsc{T-Abs}$$

$$\frac{\Gamma \vdash e \ :\ x{:}t_x \to t \qquad \Gamma \vdash e' : t_x}{\Gamma \vdash e\ e' : \exists\, x{:}t_x.\, t}\ \textsc{T-App} \qquad \frac{\Gamma \vdash e_x : t_x \quad \Gamma, x{:}t_x \vdash e_2 : t \quad \Gamma \vdash_w t}{\Gamma \vdash \texttt{let}\, x = e_x\, \texttt{in}\, e : t}\ \textsc{T-Let}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\ :\ t}\ \textsc{T-Ann} \qquad \frac{\Gamma \vdash e : s \qquad \Gamma \vdash s <: t \qquad \Gamma \vdash_w t}{\Gamma \vdash e : t}\ \textsc{T-Sub}$$

The last rule, T-Sub, uses the subtyping judgement $\Gamma \vdash s <: t$. The subtyping rules are as follows:

$$\frac{\Gamma, v_1 : b\{v_1 : p_1\} \vdash_e p_2[v_1/v_2]}{\Gamma \vdash b\{v_1 : p_1\} <: b\{v_2 : p_2\}}\ \textsc{S-Base} \qquad \frac{\Gamma \vdash s_2 <: s_1 \quad \Gamma, x_2 : s_2 \vdash t_1[x_2/x_1] <: t_2}{\Gamma \vdash x_1 : s_1 \to t_1 <: x_2 : s_2 \to t_2}\ \textsc{S-Func}$$

$$\frac{\Gamma \vdash y : t_x \qquad \Gamma \vdash t <: t'[y/x]}{\Gamma \vdash t <: \exists\, x : t_x .\, t'}\ \textsc{S-Witn} \qquad \frac{\Gamma, x : t_x \vdash t <: t' \quad x \notin \mathit{free}(t)}{\Gamma \vdash \exists\, x : t_x .\, t <: t'}\ \textsc{S-Bind}$$

The first rule above, S-Base, uses the entailment judgement $\Gamma \vdash_e c$ which states that constraint c is valid (as a first-order sentence) when universally quantified over all variables bound in environment Γ. Let Valid(c) denote the property that constraint c is a valid first-order *sentence*. We give the inference rules for the entailment judgement:

$$\frac{\textsc{Valid}(c)}{\varnothing \vdash_e c}\ \textsc{Ent-Emp} \qquad \frac{\Gamma \vdash_e \forall\, v : p \Rightarrow c}{\Gamma, v : b\{v : p\} \vdash_e c}\ \textsc{Ent-Ext}$$

2 Preliminaries

For clarity, we distinguish between different typing judgments with a subscript. The type judgments in the underlying typed lambda calculus will be denoted by $\vdash_B$ and a colon before the type. In order to speak about the base type underlying some type, we define a function that erases refinements in types:

$$\lfloor b\{x : p\} \rfloor := b, \quad \lfloor x : t_x \to t \rfloor := \lfloor t_x \rfloor \to \lfloor t \rfloor, \quad \text{and} \quad \lfloor \exists\, x : t_x .\, t \rfloor := \lfloor t \rfloor$$

We start our development of the meta-theory by giving a definition of *type denotations*. Roughly speaking, the denotation of a type t is the class of expressions e with the correct underlying base type such that if e evaluates to a value, then this value satisfies the refinement predicates that appear within the structure of t. We formalize this notion with a recursive definition:

$$\begin{aligned}
[\![b]\!] &:= \{ e \mid \varnothing \vdash_B e : b \} \\
[\![b\{x : p\}]\!] &:= \{ e \mid (\varnothing \vdash_B e : b) \wedge (\text{if}\, e \hookrightarrow^* v\, \text{then}\, p[v/x] \hookrightarrow^* \texttt{true}) \} \\
[\![x : t_x \to t]\!] &:= \{ e \mid (\varnothing \vdash_B e : \lfloor t_x \rfloor \to \lfloor t \rfloor) \wedge (\forall\, v \in [\![t_x]\!] .\, e\ v \in [\![t[v/x]]\!]) \} \\
[\![\exists\, x : t_x .\, t]\!] &:= \{ e \mid (\varnothing \vdash_B e : \lfloor t \rfloor) \wedge (\exists\, v \in [\![t_x]\!] .\, e \in [\![t[v/x]]\!]) \}
\end{aligned}$$

We also have the concept of the denotation of an environment Γ; we intuitively define this to be the set of all sequences of expression bindings for the variables in Γ such that the expressions respect the denotations of the types of the corresponding variables. A closing substitution is just a sequence of expression bindings to variables:

$$\theta = (x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) \quad \text{with all}\, x_i\, \text{distinct}$$

We use the shorthand $\theta(x)$ to refer to e_i if $x = x_i$. We define $\theta(t)$ to be the type derived from t by substituting for all variables in θ:

$$\theta(t) := t[e_1/x_1] \cdots [e_n/x_n]$$

Then we can formally define the denotation of an environment:

$$[\![\Gamma]\!] := \{\theta = (x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) \mid \forall\, (x : t) \in \Gamma.\, \theta(x) \in [\![\theta(t)]\!]\}.$$

3 Meta-theory

In this section, we seek to prove the operational soundness of our language λ_1. Our proof of the soundness theorem begins with several helping lemmas.

Lemma 1. *If $e \hookrightarrow^* e'$ then $e \in [\![t]\!]$ iff $e' \in [\![t]\!]$.*

Proof. We proceed by a case split on the definition of the denotation of a type; in other words, we use induction on the size of type t (the number of arrows or existential quantifiers appearing in t).

First, suppose that $t \equiv b$, a base type. We appeal to the soundness of the underlying bare type system. In particular, if $e \in [\![b]\!]$ then $\varnothing \vdash_B e : b$ and so $\varnothing \vdash_B e' : b$ and $e' \in [\![b]\!]$. Conversely if $e' \in [\![b]\!]$ then by determinism of the operational semantics and the typing relation, $e \in [\![b]\!]$. So we see that $e \in [\![b]\!]$. We use this argument implicitly in each of the other cases because we can replace b with any bare type.

Second, suppose $t \equiv b\{x : p\}$. If $e \in [\![t]\!]$ then it holds that if $e \hookrightarrow^* v$ for some value v then $p[v/x] \hookrightarrow^* \texttt{true}$. Suppose $e' \hookrightarrow^* v$ for some value v; then by transitive closure $e \hookrightarrow^* v$, so $p[v/x] \hookrightarrow^* \texttt{true}$ and we conclude $e' \in [\![t]\!]$.

If $e' \in [\![t]\!]$ then it holds that if $e' \hookrightarrow^* v$ for some value v then $p[v/x] \hookrightarrow^* \texttt{true}$. Suppose $e \hookrightarrow^* v$. We appeal to the determinism of the operational semantics: by hypothesis, $e \hookrightarrow^* e'$, so it must be the case that $e' \hookrightarrow^* v$. Then $p[v/x] \hookrightarrow^* \texttt{true}$ and therefore $e \in [\![t]\!]$.

Next, suppose $t \equiv x{:}t_x \to t'$. If $e \in [\![t]\!]$ then it holds that for every $v \in [\![t_x]\!]$, we have $e\, v \in [\![t'[v/x]]\!]$. Because $e \hookrightarrow^* e'$, we have $(e\, v) \hookrightarrow^* (e'\, v)$ by the operational semantics and so by induction (on the structure of t) we have $e'\, v \in [\![t'[v/x]]\!]$ and thus $e' \in [\![t]\!]$. If $e' \in [\![t]\!]$ then it holds that $\forall\, v \in [\![t_x]\!].\, e'\, v \in [\![t'[v/x]]\!]$. We appeal to the determinism of the operations semantics: by hypothesis, $e \hookrightarrow^* e'$, so it must be the case that $e\, v \hookrightarrow^* e'\, v$. Then by induction we have $e\, v \in [\![t'[v/x]]\!]$, and thus $e \in [\![t]\!]$.

Finally, suppose $t \equiv \exists\, x{:}t_x.\, t'$. We have $e \in [\![t]\!]$ if and only if there exists some $v \in [\![t_x]\!]$ such that $e \in [\![t'[v/x]]\!]$. Then by induction $e \in [\![t'[v/x]]\!]$ if and only if $e' \in [\![t'[v/x]]\!]$. By definition, $e' \in [\![t]\!]$ if and only if there exists $v \in [\![t_x]\!]$ such that $e' \in [\![t'[v/x]]\!]$, which completes the proof. □

Lemma 2. *(Type Denotations) Our typing and subtyping relations are sound with respect to the denotational semantics of our types:*
1. If $\Gamma \vdash t_1 <: t_2$ then $\forall\theta.\theta \in [\![\Gamma]\!] \Rightarrow [\![\theta(t_1)]\!] \subseteq [\![\theta(t_2)]\!]$.
2. If $\Gamma \vdash e : t$ then $\forall\theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e) \in [\![\theta(t)]\!]$.