Formalizing Simple Refinement Types in Coq

An Experience Report

Nico Lehmann
PLEIAD Lab, Computer Science Department
University of Chile
nlehmann@dcc.uchile.cl

Éric Tanter
PLEIAD Lab, Computer Science Department
University of Chile
etanter@dcc.uchile.cl

1. Overview

We consider *refinement types* in which base types can be refined by constraints expressible in some decidable logic. Such refinement types have been applied in many settings, such as certification of security policies [1, 11], and reasoning precisely about heap updates [10, 11]. A lot of work has been done on extensions of the basic idea of refinements, including inference in *liquid types* [9], or mixing static and dynamic checking with *hybrid type checking* [7]. Recently Chugh has proposed *nested refinements* [5], which allow the inclusion of type assertions in the logic of refinements. Combined with some heap reasoning, this allows static checking of complex idioms found in dynamic languages [4]. The meta-theory of these advanced forms of refinements can be tricky, and we believe it would be helpful to have a basic framework in Coq on top of which to explore more complex variants of refinement types (and their meta-theory). In this context, we hereby report on our effort to formalize in Coq a simple form of refinement types, and establish their soundness. We identify some key ideas and challenges involved in the formalization, including both the modeling of the language in Coq and the proof techniques we used.

2. Simple Refinements

In this section we present *simple refinements*, the basic system considered in our development. The syntax in Figure 1 is inspired by the presentation of Knowles and Flanagan [7], except that here the language of logical formulas is separate from program expressions.

Simple refinements have the form $\{x\!:\!B \mid p\}$, denoting the set of values of base type B for which the formula p is true. The language also includes a dependent arrow type $(x\!:\!T_1 \to T_2)$. The syntax of types is defined in three levels: types T contain formulas, and formulas p contain logical values w. As we will see, this layering has some implications for proving statements about types. Also, note that variables are considered to be values. This is necessary to have a proper interaction with the logic, as a correspondence between values in the language and constants in the logic must exist. In the logic, variables represent uninterpreted constants, which is an important feature to reason about arbitrary values that satisfy some specification.

The syntax of the logic is parametric in the set of predicates P and logical functions F. We will see later that we only make a few assumptions about the semantics of the logic. Finally, note that expressions are in A-normal form, which is important when dealing with the dependent application rule. The detailed presentation of the language is provided in Appendix for reference.

We have formalized these simple refinements in Coq[1] and proven the system to be sound. In addition to the usual *progress* and *preservation* lemmas to establish type safety, we have proven the soundness of refinements, namely that values of a refined type comply with the stated formula:

v	::=	$\lambda x.e \mid x \mid c$	Values
e	::=	$v \mid v\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e$	Terms
		$\texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e$	
T	::=	$\{x\!:\!B \mid p\} \mid x\!:\!T \to T$	Types
p, q	::=	$P(\overline{w}) \mid p \wedge q \mid p \vee q \mid \neg p$	Formulas
w	::=	$v \mid F(\overline{w})$	Logical Values

Figure 1. Simple Refinement Types Syntax

Theorem (Refinement Soundness)**.** *If* $\emptyset \vdash v :: \{x\!:\!B \mid p\}$ *then* $p[v/x]$ *is true.*

Of course these results are unsurprising: refinement types are subsumed by more expressive systems already known to be sound (e.g. Coq). We emphasize that our goal is to understand why refinement types are sound (in the spirit of Software Foundations [8]), and to provide tools to develop the metatheory of more complex refinement type systems.

3. Remarks and Challenges

We now report on some key features and design choices in our development. Some considerations are related to the modeling of simple refinements. Others are related to the particular formalization in Coq and the proofs techniques we used.

Subtyping Subtyping is key to refinement type systems. To define subtyping, we appeal to the notion of entailment or logical consequence. Following an approach similar to the one of Knowles and Flanagan [6] we parametrize the system with a judgment $\Delta \vDash p$ for which a few axioms are assumed. Here we use Δ to range over sets of formulas. In a typing derivation this context comes from the logic information extracted from a typing environment (Appendix A.4).

Subtyping simply reduces to this entailment judgment. Intuitively, in a type environment Γ, a refinement type $\{x\!:\!B \mid p\}$ is a subtype of $\{x\!:\!B \mid q\}$ if p entails q in the context extracted from Γ. Formally, we have the following rule for subtyping of refinements:

$$\frac{\mathit{extract}\,(\Gamma) \cup \{p\} \vDash q}{\Gamma \vdash \{x\!:\!B \mid p\} <: \{x\!:\!B \mid q\}}$$

Our approach differs from the one of Knowles and Flanagan in that we define explicitly a way to extract formulas from typing environments, instead of assuming a "theorem proving oracle" rep-

[1] The Coq development is available online: `https://github.com/pleiad/Refinements`.

resented by the judgment $\Gamma \vdash p \Rightarrow q$. Our definition lets us define the axioms required for the judgment by appealing only to concepts relative to the logic.

Other approaches define subtyping using *validity* [5, 9]: a query Valid $(\llbracket\Gamma\rrbracket \wedge p \rightarrow q)$ is delegated to an external solver. Here $\llbracket\Gamma\rrbracket$ corresponds to the conjunction of all the formulas extracted from Γ. This approach requires giving a specific meaning to the syntax of the logic. Moreover, a validity query is just an algorithmic way to compute the more general entailment judgment. We prefer our approach because it allow us to understand the requirements on the logic just through the meaning of the entailment judgment.

Axiomatization of the logic We want our formalization of simple refinements to be parametric, yet not any semantics for the logic will do. At least we require the logic to support means to reason about equality. We also expect the entailment judgment to respect the usual semantics, meaning that it captures the notion of a statement that logically *follows from* a set of assumptions. Finally, we require entailment to be monotone in the logic.

We encode these properties in the form of axioms about the entailment judgment. Here we state all the axioms we used to prove refinement soundness.

1. (Valid equality) $\varnothing \vDash (v = v)$.
2. (Assumption) $\Delta \cup \{p\} \vDash p$.
3. (Cut) If $\Delta \vDash p$ and $\Delta \cup p \vDash q$ then $\Delta \vDash q$.
4. (Monotonicity) If $\Delta_1 \vDash p$ then $\Delta_1 \cup \Delta_2 \vDash p$.
5. (Substitution) If $\Delta \vDash q$ then $\Delta[v/x] \vDash q[v/x]$.

The operation $\Delta[v/x]$ corresponds to the natural lifting of substitution to sets of formulas. The first three axioms correspond directly to the *natural* semantics we expect for entailment. The fourth axiom requires entailment to be monotone in the logic. The last axiom demands a little more attention. As we have dependent types we must define a substitution operation for types which is defined using the same operation for formulas. So, the operation $p[v/x]$ is inherited from the definition of substitution in types. When interpreting $p[v/x]$ in the context of an entailment judgment the operation does not directly translate to substitution of free variables in the logic. In the context of an entailment judgment x does not denote a free variable, but an uninterpreted constant. Thus, it is more accurate to say that $p[v/x]$ just gives the same name to x and v, restricting the interpretation of both constants to be the same. What makes the last axiom interesting is that v is not necessarily a fresh constant and it could be already mentioned in Δ or q. So, the axiom states that if we know that Δ entails q for any interpretation of the constants x and v we also known that the judgment holds for the specific cases where x and v have the same interpretation.

Introduction of abstractions into the logic As the language of refinements is separate from expressions we must proceed with care in the use of dependent types. The syntax of the language is defined in A-normal form so only values are allowed in the argument position of an application. Thus we only need to define substitution of values into types and we can ensure that all obligations of entailment stay within the same logic.

One more detail has to be considered though. Because values need to be introduced into the logic, the logic must be able to handle all types of values, in particular abstractions. As we are aiming for simplicity, we do not defer any reasoning about functions to the logic. Instead we introduce lambdas by assigning each one an atomic name. So, the logic can only reason about equality of lambdas using these names. We do this by requiring all lambdas to be annotated with a label, and define function equality based on these labels. Function labels could be understood as being introduced in a prior transformation of the program (e.g. along with A-normalization). Another alternative would be to pick a fresh name for each lambda introduced into the logic, thus having only reference equality. Using the labels gives flexibility, because a more interesting approach to compute equality between functions could be used.

Locally nameless representation Because the considered language has both refinements and dependent types, we must deal with bindings in both types and terms. We adopt the *locally nameless representation* [2], motivated by the large number of formal developments that adopted it successfully to simplify dealing with name binding. In brief, the locally nameless representation consists in syntactically separating bound variables (representing them with de Bruijn indices) from free variables (represented as atoms, ie. names). Notably, in the locally nameless representation, an abstraction has the form λe and to inspect the body e, one has to first provide a fresh name x to *open* the abstraction. The *variable opening* operation, written e^x, replaces all the indices of e bound to the abstraction λe with the name x.

Often we have to define inference rules that hold only for variables that are "fresh enough"; of course we need to precisely define this notion. Following Charguéraud, we combine the locally nameless representation with *cofinite quantification*, which consists in existentially quantifying over a finite set of names L for which the given rule does not hold. Thus the rule applies to the cofinitely-many names not in L. As an elimination form the cofinite rule offers a strong hypothesis because the property holds for cofinitely-many names. As an introduction form it is easy to find a set L for which the rule holds and in most cases involves choosing L as large as possible in the context of a proof.

As a drawback, it is necessary to define substitution for free variables besides the already mentioned opening operation. It is then necessary to prove a number of lemmas relating both operations. In the case of simple refinements the number of lemmas is duplicated because the same properties must be proven for types and terms. However, these lemmas are easy to prove and once proven, the rest of the proofs proceeded with ease in this respect.

Weakening Our proofs also present a lot of obligations that can be resolved just by weakening a hypothesis. Weakening lemmas have to be proved and stated in a general way, allowing the weakening of the environment arbitrarily in the middle. For example the weakening lemma for the typing judgment has the following form:

$$\frac{\vdash \Gamma_1, \Gamma_2, \Gamma_3 \qquad \Gamma_1, \Gamma_3 \vdash e :: T}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash e :: T}$$

This makes it impossible to apply the lemmas directly and environments have to be rewritten to match the actual statements. We have developed some tactics that try to "guess" the needed rewriting based on the environment in the current goal and the environment in the hypothesis. This allowed us to automatize most of the proofs that can be directly resolved by weakening some hypothesis. For example, our tactics can automatically prove the judgment $\Gamma_1, x\!:\!T, \Gamma_2 \vdash e :: T$ in a context with the hypothesis $\emptyset \vdash e :: T$.

Well-formedness For an environment to be well-formed all free variables in types must be defined upfront (Appendix A.2). The typing and subtyping judgments are defined so they hold only for well-formed objects. A main issue in our development was the proof of this property. However, most obligations of well-formedness can be directly resolved by extracting properties and inverting hypotheses in the context. We have automatized this task and most obligations of well-formedness are resolved automatically. We adapted this idea from the examples found in the locally nameless library[2].

[2] `http://www.chargueraud.org/softs/ln/`

Combined induction principle Finally, probably the most interesting technique we came up with is for dealing with the three levels of syntax in types mentioned in Section 2. Most properties about types involve making a proof of the same property lifted to formulas, and then to logical values. Despite not being necessary, for convenience, we define types, formulas and logical values to be mutually recursive. This "trick" allows us to easily define a combined mutual induction principle and make proofs about the three syntactic levels at the same time. We end up with a series of tuples with properties about the three levels. This technique turned out to be really convenient because in most cases the `apply` tactic can automatically decide which version of the property to use.

4. Perspective

The formalization of simple refinements presented some difficulties mostly because of the structure of types (e.g. free variables in the well-formedness definition and three-level structure of types). We developed some proof techniques to address these points more conveniently. We believe these insights can be used in the formalization of similar systems.

Based on this preliminary work we now want to explore more complex refinement type systems. In particular, Knowles and Flanagan proposed the use of existential types to recover compositional reasoning in refinement types [6]. Additionally, this system supports the definition of an algorithmic typing judgment, as used in Chugh's implementation of nested refinements [3]. We are currently exploring these ideas in Coq on top of the simple refinements presented here.

References

[1] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8:1–8:45, Jan. 2011.

[2] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011.

[3] R. Chugh. *Nested Refinement Types for JavaScript*. PhD thesis, University of California, Sept. 2013.

[4] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*, pages 587–606, Tucson, AZ, USA, Oct. 2012. ACM Press.

[5] R. Chugh, P. M. Rondon, A. Bakst, and R. Jhala. Nested refinements: a logic for duck typing. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 231–244, Philadelphia, USA, Jan. 2012. ACM Press.

[6] K. Knowles and C. Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 27–38, New York, NY, USA, 2008. ACM.

[7] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):Article n.6, Jan. 2010.

[8] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2015. http://www.cis.upenn.edu/ bcpierce/sf.

[9] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 159–169. ACM Press, June 2008.

[10] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 131–144, New York, NY, USA, 2010. ACM.

[11] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *ESOP 2010: 19th European Symposium on Programming*, number MSR-TR-2009-164. Springer Verlag, March 2010.

A. Appendix

A.1 Syntax

$$c \in \textbf{Constants}, \quad x \in \textbf{ValueIdentifiers}$$
$$P \in \textbf{PredicateSymbols}, \quad F \in \textbf{FunctionSymbols}$$

v	$::=$	$\lambda x.e \mid x \mid c$	Values
e	$::=$	$v \mid v\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e$	Terms
		$\texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e$	
T	$::=$	$\{x\!:\!B \mid p\} \mid x\!:\!T \to T$	Types
p, q	$::=$	$P(\overline{w}) \mid p \wedge q \mid p \vee q \mid \neg p$	Formulas
w	$::=$	$v \mid F(\overline{w})$	Logical Values
Γ	$::=$	$\emptyset \mid \Gamma, x\!:\!T \mid \Gamma, p$	Environments

A.2 Well-formedness

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Gamma \quad fv(p) \subseteq dom(\Gamma)}{\vdash \Gamma, p}$$

$$\frac{\vdash \Gamma \quad x \notin dom(\Gamma) \quad fv(T) \subseteq dom(\Gamma)}{\vdash \Gamma, x\!:\!T}$$

$\boxed{\Gamma \vdash T}$

$$\frac{\vdash \Gamma \quad fv(T) \subseteq dom(\Gamma)}{\Gamma \vdash T}$$

A.3 Typing judgment

$\boxed{\Gamma \vdash e :: T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash c :: ty(c)}$$

$$\frac{\vdash \Gamma \quad \Gamma(x) = \{y\!:\!B \mid p\}}{\Gamma \vdash x :: \{z\!:\!B \mid z = y\}}$$

$$\frac{\vdash \Gamma \quad \Gamma(x) = y\!:\!T_1 \to T_2}{\Gamma \vdash x :: y\!:\!T_1 \to T_2}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x\!:\!T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.e :: x\!:\!T_1 \to T_2}$$

$$\frac{\Gamma \vdash v_1 :: x\!:\!T_1 \to T_2 \quad \Gamma \vdash v_2 :: T_1}{\Gamma \vdash v_1\ v_2 :: T_2[v_2/x]}$$

$$\frac{\begin{array}{c}\Gamma \vdash v :: \{x\!:\!\texttt{Bool} \mid p\} \\ \Gamma, v = \texttt{true} \vdash e_1 :: T \quad \Gamma, v = \texttt{false} \vdash e_2 :: T\end{array}}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: T}$$

$$\frac{\Gamma \vdash e_1 :: T_1 \quad \Gamma, x\!:\!T_1 \vdash e_2 :: T_2 \quad \Gamma \vdash T_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: T_2}$$

$$\frac{\Gamma \vdash e :: T_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash e :: T_2}$$

A.4 Subtyping

$\boxed{\Gamma \vdash e <: T}$

$$\frac{\begin{array}{c}\mathit{extract}\,(\Gamma) \cup \{p\} \vDash q \\ \Gamma \vdash \{x\!:\!B \mid p\} \quad \Gamma \vdash \{x\!:\!B \mid q\}\end{array}}{\Gamma \vdash \{x\!:\!B \mid p\} <: \{x\!:\!B \mid q\}}$$

$$\frac{\Gamma \vdash T_{21} <: T_{11} \quad \Gamma, x\!:\!T_{21} \vdash T_{12} <: T_{22}}{\Gamma \vdash x\!:\!T_{11} \to T_{12} <: x\!:\!T_{21} \to T_{22}}$$

$$
\begin{aligned}
\mathit{extract}\,(\emptyset) &= \varnothing \\
\mathit{extract}\,(\Gamma, x : \{y : B \mid p\}) &= \mathit{extract}\,(\Gamma) \cup p[x/y] \\
\mathit{extract}\,(\Gamma, p) &= \mathit{extract}\,(\Gamma) \cup \{p\} \\
\mathit{extract}\,(\Gamma, x : T_1 \to T_2) &= \mathit{extract}\,(\Gamma)
\end{aligned}
$$

A.5 Soundness

Soundness of the system is expressed in the three following properties.

Theorem 1 (Progress)**.** *If* $\emptyset \vdash e :: T$ *then either exists some* e' *such that* $e \to e'$ *or* e *is a value.*

Theorem 2 (Preservation)**.** *If* $\emptyset \vdash e :: T$ *and* $e \to e'$ *then* $\emptyset \vdash e' :: T$.

Theorem 3 (Refinement Soundness)**.** *If* $\emptyset \vdash v :: \{x : B \mid p\}$ *then* $\emptyset \vDash p[v/x]$.