Refinement Types and Algebraic Effects

Danel Ahman

LFCS, School of Informatics, University of Edinburgh

1. Introduction

In software development, programs and software components are often intended to behave according to some pre-given specifications, usually created at the analysis stage. However, widespread programming languages and their type systems are generally too weak to encode and enforce such specifications. To address this issue, various approaches have been developed to extend standard type systems to support different forms of specifications, some examples being session types, refinement types, contracts, ownership types, Hoare Type Theory and dependent types. All of these systems give at least one important guarantee: whenever programs are well-formed and well-typed, they adhere to the encoded specifications. This guarantee is either enforced at compile time during type-checking or by a suitable run-time environment. These methods behave well for their specialist problem domains, e.g.: session types for specifying network communication behavior and Hoare Type Theory for specifying pre- and post-conditions on state manipulations. However, as each of these methods usually only accounts for a very particular set of specifications and notions of computation, extending the underlying programming language with new features also requires the method under question to be modified and its properties re-proved.

To remedy the above deficiencies, we propose to study a more general framework that could account for a wide range of different notions of computation and logical specification. Our proposal is to take refinement types as a basis and combine them with Levy's Call-by-Push-Value (CBPV) [4] paradigm to account for both properties on values that programs manipulate and for properties on the effectful behavior that the programs are intended to exhibit. To accommodate different notions of computation, such as state, exceptions, input/output and probabilistic computations, we propose to build our work on the algebraic treatment of computational effects of Plotkin and Power [6].

Due to lack of space we only give a very brief overview of our preliminary work. More details can be found in the author's PhD thesis proposal[1].

Acknowledgments

I am very grateful to my PhD supervisors Gordon Plotkin and Alex Simpson for helpful discussions. I also thank Sam Lindley and Ben Kavanagh for useful discussions.

2. A refinement type system for CBPV

Our refinement type system is inspired by the theoretical treatment given by Denney [1]. In particular, we follow Denney's lead in using indexed kinds to track the underlying refinement-free CBPV types. As we base our work on CBPV, we can exploit its separation between values and computations and define well-formed *value* and *computation refinement types* in Figure 1 by using two judgments: $\vdash \sigma : \mathsf{Ref}(A)$ and $\vdash \underline{\tau} : \underline{\mathsf{Ref}}(\underline{C})$. The kind indices A and $\underline{C}$ are the underlying CBPV value and computation types. Most of the rules are standard from CBPV, modulo the indexed kinds. The important rules are the last two which introduce logical specifications into the type system. These specifications φ are defined as formulas in the logic of algebraic effects introduced by Plotkin and Pretnar [7].

The reader should note that here we present a type system in which refinement types do not depend on the free variables as opposed to the context-dependent refinement types appearing in Denney's work and also, for example, in the F* project [8]. This choice was made to start with a simpler semantic analysis. We later intend to extend the type system with additional context dependence.

We accompany our definitions with an operation $\lceil - \rceil$ on types and terms that forgets the extra structure of the refinement type system and returns the underlying CBPV types and terms, e.g.: $\lceil \vdash \sigma : \mathsf{Ref}(A) \rceil \overset{\text{def}}{=} A$. As we will overload notation and use the CBPV term constructor names also in our type system, $\lceil - \rceil$ acts on term constructors by identity, e.g.: $\lceil \mathsf{proj}_i\, V \rceil \overset{\text{def}}{=} \mathsf{proj}_i \lceil V \rceil$. In addition, we also define an operation $(-)^\bullet$ that translates refinement types $\sigma, \underline{\tau}$ to corresponding propositions $x : \lceil \sigma \rceil \vdash \sigma^\bullet$ and $\zeta : \lceil \underline{\tau} \rceil \vdash \underline{\tau}^\bullet$ in the logic of algebraic effects. The definition of $(-)^\bullet$ is given by mutual structural recursion on both value and computation refinement types. Due to space constraints we only illustrate the definition for $\vdash \{x : \sigma \mid \varphi\} : \mathsf{Ref}(A)$ and $\vdash F\sigma : \underline{\mathsf{Ref}}(FA)$:

$$
\begin{aligned}
(\{x' : \sigma \mid \varphi\})^\bullet &\overset{\text{def}}{=} \varphi[x/x'] \wedge \sigma^\bullet \\
(F\sigma)^\bullet &\overset{\text{def}}{=} (\mu X.((\zeta : \lceil F\sigma \rceil).(\exists x : \lceil \sigma \rceil . \zeta \equiv \mathsf{return}\, x \wedge \\
&\qquad\qquad \sigma^\bullet(x)) \vee (\langle - \rangle (X)(\zeta))))(\zeta)
\end{aligned}
$$

Intuitively, the proposition $\zeta : \lceil F\sigma \rceil \vdash (F\sigma)^\bullet$ is satisfied by those computations whose return values satisfy $x : \lceil \sigma \rceil \vdash \sigma^\bullet$.

The terms of our language are divided into values and computations, similarly to CBPV. The well-typed terms are mutually defined in Figure 2 and given by judgments $\Gamma \vdash_{\mathsf{v}} V : \sigma$ and $\Gamma \vdash_{\mathsf{c}} M : \underline{\tau}$. The first six rules define the introduction and elimination principles for the refinements $\{x : \sigma \mid \varphi\}$ and $\{\zeta : \underline{\tau} \mid \varphi\}$. The rules on the second and third rows present the variables and terms of value unit type, value product type, and value coproduct type. In addition we also have term constructors f corresponding to the function symbols in the base signature of the effect theories of Plotkin and Pretnar [7]. In the case of a natural numbers base type nat, fs could be addition, multiplication, etc. The third row presents the thunking and forcing constructs that turn computations into values and vice versa. The fourth row describes terms of computation product type, the free computation type and function type.

The last two rows present typing rules for the monadic sequencing and algebraic effect operations. While most of the previous rules are familiar from CBPV, these rules are a little bit different. Recall that in CBPV, the operations $\mathsf{op} : \beta ; \alpha$ are typed with the following rule:

$$
\frac{\Gamma \vdash_{\mathsf{v}} V : \beta \qquad \Gamma, x : \alpha \vdash_{\mathsf{c}} M : \underline{C}}{\Gamma \vdash_{\mathsf{c}} \mathsf{op}_V((x : \alpha).M) : \underline{C}}
$$

However, in the presence of refinement types, this rule will in general not be sound. The intuitive cause of unsoundness is that the refinement type σ at which we are typing $\mathsf{op}_V((x : \alpha).M)$ might, for example, assert that the specific op is not allowed to inhabit σ. Moreover, as semantically operations on product models are defined component-wise, operations at product refinement types will only be well-typed if they are well-typed at both components. Similar also holds for typing operations at function type. The monadic sequencing $M\, \mathsf{to}\, x.\, N$ has to also be typed in this type-directed manner.

Finally, it is also straightforward to define refinement relations for both value and computation refinement types: $\Gamma \vdash_{\mathsf{r}} \sigma_2 \sqsubseteq \sigma_1$ and $\Gamma \vdash_{\mathsf{r}} \underline{\tau}_2 \sqsubseteq \underline{\tau}_1$. Based on these refinement relations, we can also define weakening principles for values and computations:

$$
\frac{\Gamma \vdash_{\mathsf{v}} V : \sigma_2 \qquad \Gamma \vdash_{\mathsf{r}} \sigma_2 \sqsubseteq \sigma_1}{\Gamma \vdash_{\mathsf{v}} V : \sigma_1}
\qquad
\frac{\Gamma \vdash_{\mathsf{c}} M : \underline{\tau}_2 \qquad \Gamma \vdash_{\mathsf{r}} \underline{\tau}_2 \sqsubseteq \underline{\tau}_1}{\Gamma \vdash_{\mathsf{c}} M : \underline{\tau}_1}
$$

$$
\frac{}{\vdash \alpha : \mathsf{Ref}(\alpha)} \qquad \frac{}{\vdash 1 : \mathsf{Ref}(1)} \qquad \frac{}{\vdash 0 : \mathsf{Ref}(0)}
$$

$$
\frac{\vdash \underline{\tau} : \underline{\mathsf{Ref}}(\underline{C})}{\vdash U\underline{\tau} : \mathsf{Ref}(U\underline{C})} \qquad \frac{\vdash \sigma : \mathsf{Ref}(A)}{\vdash F\sigma : \underline{\mathsf{Ref}}(FA)}
$$

$$
\frac{\vdash \sigma_i : \mathsf{Ref}(A_i) \quad (i \in \{1,2\})}{\vdash \sigma_1 + \sigma_2 : \mathsf{Ref}(A_1 + A_2)} \qquad \frac{\vdash \underline{\tau}_i : \underline{\mathsf{Ref}}(\underline{C}_i) \quad (i \in \{1,2\})}{\vdash \underline{\tau}_1 \times \underline{\tau}_2 : \underline{\mathsf{Ref}}(\underline{C}_1 \times \underline{C}_2)}
$$

$$
\frac{\vdash \sigma_1 : \mathsf{Ref}(A_1) \quad \vdash \sigma_2 : \mathsf{Ref}(A_2)}{\vdash \sigma_1 \times \sigma_2 : \mathsf{Ref}(A_1 \times A_2)} \qquad \frac{\vdash \sigma : \mathsf{Ref}(A) \quad \vdash \underline{\tau} : \underline{\mathsf{Ref}}(\underline{C})}{\vdash \sigma \to \underline{\tau} : \underline{\mathsf{Ref}}(A \to \underline{C})}
$$

$$
\frac{\vdash \sigma : \mathsf{Ref}(A) \quad x : A \vdash \varphi : \mathsf{prop}}{\vdash \{x : \sigma \mid \varphi\} : \mathsf{Ref}(A)} \qquad \frac{\vdash \underline{\tau} : \underline{\mathsf{Ref}}(\underline{C}) \quad \zeta : \underline{C} \vdash \varphi : \mathsf{prop}}{\vdash \{\zeta : \underline{\tau} \mid \varphi\} : \underline{\mathsf{Ref}}(\underline{C})}
$$

Figure 1. Well-formed value and computation refinement types

3. Semantics

We sketch a concrete categorical semantics for our refinement type system using the subobject fibration $\mathsf{Sub}(\mathsf{Set}) \to \mathsf{Set}$. We intend

[1] Author's PhD thesis proposal:
http://homepages.inf.ed.ac.uk/s1225336/papers/thesisproposal.pdf

$$\frac{\Gamma \vDash_v V : \sigma \quad \lceil\Gamma\rceil \mid \Gamma^{\bullet} \vdash \varphi[\lceil V\rceil/x]}{\Gamma \vDash_v V : \{x : \sigma \mid \varphi\}} \qquad \frac{\Gamma \vDash_v V : \{x : \sigma \mid \varphi\}}{\Gamma \vDash_v V : \sigma} \qquad \frac{\Gamma \vDash_v V : \{x : \sigma \mid \varphi\}}{\lceil\Gamma\rceil \mid \Gamma^{\bullet} \vdash \varphi[\lceil V\rceil/x]} \qquad \frac{\Gamma \vDash_c M : \underline{\tau} \quad \lceil\Gamma\rceil \mid \Gamma^{\bullet} \vdash \varphi[\lceil M\rceil/x]}{\Gamma \vDash_c M : \{\zeta : \underline{\tau} \mid \varphi\}} \qquad \frac{\Gamma \vDash_c M : \{\zeta : \underline{\tau} \mid \varphi\}}{\Gamma \vDash_c M : \underline{\tau}}$$

$$\frac{\Gamma \vDash_c M : \{\zeta : \underline{\tau} \mid \varphi\}}{\lceil\Gamma\rceil \mid \Gamma^{\bullet} \vdash \varphi[\lceil M\rceil/\zeta]} \qquad \frac{\vdash \Gamma, x : \sigma, \Gamma' \ \mathsf{wf}}{\Gamma, x : \sigma, \Gamma' \vDash_v x : \sigma} \qquad \frac{\vdash \Gamma \ \mathsf{wf}}{\Gamma \vDash_v \star : 1} \qquad \frac{\Gamma \vDash_v V_1 : \beta_1 \quad \dots \quad \Gamma \vDash_v V_n : \beta_n}{\Gamma \vDash_v f(V_1, \dots, V_n) : \beta} \qquad \frac{\Gamma \vDash_v V : \sigma_1 \quad \Gamma \vDash_v W : \sigma_2}{\Gamma \vDash_v \langle V, W\rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vDash_v V : \sigma_1 \times \sigma_2}{\Gamma \vDash_v \mathsf{proj}_i\, V : \sigma_i}$$

$$\frac{\Gamma \vDash_v V : \sigma_i}{\Gamma \vDash_v \mathsf{inj}_i\, V : \sigma_1 + \sigma_2} \qquad \frac{\Gamma \vDash_v V : 0}{\Gamma \vDash_v \mathsf{pmatch}\, V \,\mathsf{as}\, \{\} : \sigma} \qquad \frac{\Gamma \vDash_v V : \sigma_1 + \sigma_2 \quad \Gamma, x_i : \sigma_i, y : \{x : 1 \mid \lceil V\rceil \equiv \mathsf{inj}_i\, x_i\} \vDash_v W_i : \sigma}{\Gamma \vDash_v \mathsf{pmatch}\, V \,\mathsf{as}\, \{\mathsf{inj}_1\,(x_1) \mapsto W_1, \mathsf{inj}_2\,(x_2) \mapsto W_2\} : \sigma} \qquad \frac{\Gamma \vDash_c M : \underline{\tau}}{\Gamma \vDash_v \mathsf{thunk}\, M : U\underline{\tau}} \qquad \frac{\Gamma \vDash_v V : U\underline{\tau}}{\Gamma \vDash_c \mathsf{force}\, V : \underline{\tau}}$$

$$\frac{\Gamma \vDash_c M_1 : \underline{\tau}_1 \quad \Gamma \vDash_c M_2 : \underline{\tau}_2}{\Gamma \vDash_c \langle M_1, M_2\rangle : \underline{\tau}_1 \times \underline{\tau}_2} \qquad \frac{\Gamma \vDash_c M : \underline{\tau}_1 \times \underline{\tau}_2}{\Gamma \vDash_c \mathsf{proj}_i\, M : \underline{\tau}_i} \qquad \frac{\Gamma \vDash_v V : \sigma}{\Gamma \vDash_c \mathsf{return}\, V : F\sigma} \qquad \frac{\Gamma \vDash_c M : \sigma \to \underline{\tau} \quad \Gamma \vDash_v V : \sigma}{\Gamma \vDash_c MV : \underline{\tau}} \qquad \frac{\Gamma, x : \sigma \vDash_c M : \underline{\tau}}{\Gamma \vDash_c \lambda x : \sigma.M : \sigma \to \underline{\tau}}$$

$$\frac{\Gamma \vDash_c M : F\sigma_1 \quad \Gamma, x : \sigma_1 \vDash_c N : F\sigma_2}{\Gamma \vDash_c M \,\mathsf{to}\, x.\, N : F\sigma_2} \qquad \frac{\Gamma \vDash_c M \,\mathsf{to}\, x.\, (\mathsf{fst}\, N) : \underline{\tau}_1 \quad \Gamma \vDash_c M \,\mathsf{to}\, x.\, (\mathsf{snd}\, N) : \underline{\tau}_2}{\Gamma \vDash_c M \,\mathsf{to}\, x.\, N : \underline{\tau}_1 \times \underline{\tau}_2} \qquad \frac{\Gamma, y : \sigma \vDash_c M \,\mathsf{to}\, x.\, (Ny) : \underline{\tau}}{\Gamma \vDash_c M \,\mathsf{to}\, x.\, N : \sigma \to \underline{\tau}}$$

$$\frac{\Gamma \vDash_v V : \beta \quad \Gamma, x : \alpha \vDash_c M : F\sigma}{\Gamma \vDash_c \mathsf{op}_V((x : \alpha).M) : F\sigma} \qquad \frac{\Gamma \vDash_c \mathsf{op}_V((x : \alpha).(\mathsf{fst}\, M)) : \underline{\tau}_1 \quad \Gamma \vDash_c \mathsf{op}_V((x : \alpha).(\mathsf{snd}\, M)) : \underline{\tau}_2}{\Gamma \vDash_c \mathsf{op}_V((x : \alpha).M) : \underline{\tau}_1 \times \underline{\tau}_2} \qquad \frac{\Gamma, y : \sigma \vDash_c \mathsf{op}_V((x : \alpha).(My)) : \underline{\tau}}{\Gamma \vDash_c \mathsf{op}_V((x : \alpha).M) : \sigma \to \underline{\tau}}$$

Figure 2. Well-typed value and computation terms

to investigate more general and abstract classes of models for our refinement type system in the near future. We decided to use a subobject fibration because it allows us to model the intuitive reading of refinement types. Namely, refinement types, as they are defined above, should be semantically understood as pairs of an underlying type and a predicate over it, i.e., as a pair of a set and its subset.

More specifically, we build our semantics on top of the standard semantics for CBPV and the logic of algebraic effects in Set. To remind the reader, a CBPV model is usually given by an adjunction $F \dashv U$ between Set and a category of algebras over Set. CBPV value types are interpreted as objects $[\![A]\!]$ in Set and computation types as algebras $[\![\underline{C}]\!]$. Value terms are interpreted as morphisms $[\![\Gamma \vDash_v V : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$ and computation terms are interpreted as morphisms $[\![\Gamma \vDash_c M : \underline{C}]\!] : [\![\Gamma]\!] \to U[\![\underline{C}]\!]$. Plotkin and Pretnar give the logic of algebraic effects a standard Set-based semantics: propositions $\Gamma; \Delta; \Pi \vdash \varphi : \mathsf{prop}$ are interpreted as subsets $[\![\varphi]\!] \subseteq [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!]$ and predicates $\Gamma; \Delta; \Pi \vdash \pi : (\vec{A}, \vec{\underline{C}}) \to \mathsf{prop}$ are interpreted as maps $[\![\pi]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!] \to \mathcal{P}([\![\vec{A}]\!] \times U[\![\vec{\underline{C}}]\!])$. For our purposes, we consider this semantics in a more general fibred setting.

We interpret value and computation refinement types as objects in $\mathsf{Sub}(\mathsf{Set})$: $[\![\vdash \sigma : \mathsf{Ref}(A)]\!] \overset{\text{def}}{=} [\![\sigma]\!] \rightarrowtail [\![A]\!]$ and $[\![\vdash \underline{\tau} : \underline{\mathsf{Ref}}(\underline{C})]\!] \overset{\text{def}}{=} [\![\underline{\tau}]\!] \rightarrowtail U[\![\underline{C}]\!]$, i.e., as subobjects of $[\![A]\!]$ and $U[\![\underline{C}]\!]$. Value and computation terms are interpreted as morphisms in $\mathsf{Sub}(\mathsf{Set})$. In particular, a morphism between $[\![\Gamma]\!] \rightarrowtail [\![\lceil\Gamma\rceil]\!]$ and $[\![\sigma]\!] \rightarrowtail [\![A]\!]$ is given by a Set morphism between $[\![\lceil\Gamma\rceil]\!]$ and $[\![A]\!]$ together with a necessarily unique Set morphism between $[\![\Gamma]\!]$ and $[\![\sigma]\!]$ commuting with the subobjects. This definition of a morphism again follows the intuitive reading of refinement types: the underlying morphism between $[\![\lceil\Gamma\rceil]\!]$ and $[\![A]\!]$ denotes a CBPV value term whilst the existence of a morphism between $[\![\Gamma]\!]$ and $[\![\sigma]\!]$ ensures that the term obeys its refinements. Computation terms are interpreted similarly. A more thorough, diagrammatical, explanation of this subobject fibration based semantics can be again found in author's PhD thesis proposal.

Finally, it is important to notice that our semantics for refinement types differs from the semantics considered by Jacobs [3]. In particular, while he interprets a refinement type directly as a subset, using the notion of comprehension, we interpret it as a pair of an underlying set and a subset. However, we expect that Jacobs's work will provide us with inspiration for developing more general classes of models for our refinement type system.

4. Example specifications

We conclude by briefly discussing how one might account for both Hoare Type Theory [5] style pre- and post-condition specifications and session type [2] style network communication specifications. We also briefly discuss how our approach allows one to combine different kinds of specifications and different notions of computation.

4.1 Hoare Type Theory style pre- and post-conditions

We start with the theory of global state given by the operations lookup, update and equations as proposed by Plotkin and Power [6]. We can express the Hoare type $\{P\}x : A\{Q\}$ as a computation refinement type $\vdash \{\zeta : FA \mid P \blacktriangleright_{x:A} Q\} : \underline{\mathsf{Ref}}(FA)$ with the proposition $P \blacktriangleright_{x:A} Q$ defined as:

$$\begin{aligned}
&\zeta : FA \vdash P \blacktriangleright_{x:A} Q \overset{\text{def}}{=} \\
&\quad \forall x_{l_1} : \mathsf{int}, \dots, x_{l_n} : \mathsf{int}, y_{l_1} : \mathsf{int}, \dots, y_{l_n} : \mathsf{int}, z : A\,. \\
&\qquad P^{\blacktriangleright}[x_{l_1}/x_1, \dots, x_{l_n}/x_n] \wedge \\
&\qquad\quad \exists \zeta' : F1\,.\, \mathsf{update}_{l_1, x_{l_1}}(\ \dots\ (\mathsf{update}_{l_n, x_{l_n}}(\zeta))) \equiv \\
&\qquad\quad \zeta' \,\mathsf{to}\, x.\, \mathsf{update}_{l_1, y_{l_1}}(\ \dots\ (\mathsf{update}_{l_n, y_{l_n}}(\mathsf{return}\, z))) \\
&\qquad \Longrightarrow\ Q^{\blacktriangleright}[y_{l_1}/x_1, \dots, y_{l_n}/x_n, z/x]
\end{aligned}$$

This refinement type makes the partial correctness argument explicit. Whenever a program, when started in a state satisfying P, terminates in some state, then that final state together with the return value of type A has to satisfy Q.

4.2 Session types style network communication specifications

We start with the theory of input/output given by the operations $\mathsf{receive}$, send with no equations. By taking inspiration from session types, we can consider the following simple grammar for session refinements, i.e., specifications on how processes should send and receive bits, i.e.: $S_i ::= end_i \mid !\mathsf{bit}.S_i \mid ?\mathsf{bit}.S_i$. Notice, that we equip the process types with session specifications rather than the channel types. More formally, we define end_i as a predicate in the logic of algebraic effects and $!\mathsf{bit}.(-), ?\mathsf{bit}.(-)$ as operations on predicates. Then, a process $\Gamma \vDash_c M : \{\zeta : F1 \mid S_i(\zeta)\}$ uses channel i exactly as specified by S_i.

4.3 Combining different specifications

Although we do not show it explicitly here, we can combine both kinds of specifications by combining the theories of state and input/output. By using this composite theory, we can define computation refinement types which both specify how a process should communicate over the network and also how a process should manipulate the state. Moreover, we can extend this analysis with other algebraic effects as well, e.g., exceptions, non-determinism, probabilistic computations, etc.

5. Conclusion

We only present very preliminary results. There is still a long way to go for a satisfactory theory. For example, we do not account for local state and we have not integrated the explicit linearity of network communication found in the session type literature.

References

[1] E. Denney. *A Theory of Program Refinement*. PhD thesis, University of Edinburgh, 1998.

[2] K. Honda. Types for dyadic interaction. In *Proc. CONCUR'93*, pages 509–523, 1993.

[3] B. Jacobs. *Categorical Logic and Type Theory*. North Holland, Amsterdam, 1999.

[4] P. B. Levy. *Call-by-Push-Value. A Functional/Imperative Synthesis*. Springer, 2004.

[5] A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.

[6] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FOSSACS'02*, pages 342–356, 2002.

[7] M. Pretnar. *The logic and handling of algebraic effects*. PhD thesis, University of Edinburgh, 2010.

[8] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proc. ICFP'11*, pages 266–278, 2011.