Refine your types!

Romain Ruetschi

31.05.2018

Two words about me

Hi, my name is Romain Ruetschi, but you can also call me Romac.

I am usually found online under various spellings of “romac”.

Twitter:	https://twitter.com/_romac
GitHub:	https://github.com/romac
Homepage:	https://romac.me

Refinement types

Refinement types

1. A quick introduction
2. Under the hood
3. In the wild

A quick introduction

Types

`Int`

Types

`Int`

`Boolean`

Types

`Int`

`Boolean`

`List[Int]`

Types

`Int`

`Boolean`

`List[Int]`

`List[A]` $\rightarrow$ `Int`

Types

3 : `Int`

Types

3 : `Int`

`true` : `Boolean`

Types

$3 : \texttt{Int}$

$\texttt{true} : \texttt{Boolean}$

$\texttt{List}(1, 2, 3) : \texttt{List[Int]}$

Types

$3 : \texttt{Int}$

$\texttt{true} : \texttt{Boolean}$

$\texttt{List}(1, 2, 3) : \texttt{List[Int]}$

$\texttt{(xs: List[A]) => xs.length} : \texttt{List[A]} \to \texttt{Int}$

Refinement types

$$\{ \, x : T \mid p \, \}$$

Refinement types

$$\{\ n : \texttt{Int} \mid n > 0\ \}$$

Refinement types

$$\{\ n : \texttt{Int} \mid n > 0\ \}$$

$$\{\ xs : \texttt{List[A]} \mid \texttt{!}xs\texttt{.isEmpty}\ \}$$

Refinement types

$$\{\ n : \texttt{Int} \mid n > 0\ \}$$

$$\{\ xs : \texttt{List[A]} \mid \texttt{!}xs\texttt{.isEmpty}\ \}$$

$$\{\ t : \texttt{Tree} \mid \texttt{isBalanced(}t\texttt{)}\ \}$$

Refinement types

$$\{\ xs : \texttt{List[A]} \mid xs\texttt{.length} \neq 0\} \rightarrow A$$

Refinement types

$$\{\ xs : \texttt{List[A]} \mid xs\texttt{.length} \neq 0\} \rightarrow A$$

$$\texttt{List[A]} \rightarrow \{\ n : \texttt{Int} \mid n \geq 0\}$$

Refinement types

$$A \to \{\ n : \texttt{Int} \mid x \geq 0\} \to \{\ xs : \texttt{List[A]} \mid xs\texttt{.length} = n\ \}$$

Relation with dependent types

Not easy to precisely define either system. One view is that:

- With dependent types, types can refer to terms, the calculus is normalizing.
- With refinement types, types don't necessarily need to be able to refer to terms, and the calculus does not need to be normalizing, because proofs are discharged to a *solver*.
- In practice, it is natural to allow refinement types to refer to terms.

Relation with dependent types, continued

If we restrict ourselves to the view that *dependent types* $\approx$ *Coq* and *refinement types* $\approx$ *LiquidHaskell*, then:

- Dependent types are more expressive than refinement types, ie. one can model pretty much any kind of mathematics using dependent types, tactics, and manual proofs.
- Refinement types are more suited for automation, as predicates are drawn from a decidable logic, and proof obligations can thus be discharged to an SMT solver.

Under the hood

Subtyping

Refinement types rest on the following notion of *subtyping*:

$$\Gamma \vdash \{\ x : A \mid p\ \} \preceq \{\ y : A \mid q\ \}$$

$$\Leftrightarrow$$

$$\mathit{Valid}\Big(\llbracket \Gamma \rrbracket \wedge \llbracket p \rrbracket \Rightarrow \llbracket q \rrbracket\Big)$$

$$\Leftrightarrow$$

$$\mathit{CheckSat}\Big(\neg(\llbracket \Gamma \rrbracket \wedge \llbracket p \rrbracket \Rightarrow \llbracket q \rrbracket)\Big) = \mathit{UNSAT}$$

Subtyping (example)

$$\{\,\texttt{val x: Int = 42 }\}\vdash \texttt{\{ y: Int | y > x \}} \preceq \texttt{\{ z: Int | z > 0 \}}$$

$$\Leftrightarrow$$

$$\mathit{Valid}\big((x = 42 \ \wedge\ y > x \ \wedge\ z = y) \ \Rightarrow\ z > 0\big)$$

$$\Leftrightarrow$$

$$\mathit{CheckSat}\Big(\neg((x = 42 \ \wedge\ y > x \ \wedge\ z = y) \ \Rightarrow\ z > 0)\Big) = \mathit{UNSAT}$$

Contracts

This function

```
def f(x: Int { x > 0 }): { y: Int | y < 0 } = 0 - x
```

is correct if

$$\{\,\texttt{x: Int | x > 0}\,\} \vdash \{\,\texttt{z: Int | z = 0 - x}\,\} \preceq \{\,\texttt{y: Int | y < 0}\,\}$$

$$\Leftrightarrow$$

$$\mathit{Valid}\Big((x > 0 \ \wedge\ (z = 0 - x)\ \wedge\ (y = z)) \ \Rightarrow\ y < 0\Big)$$

Solving constraints with SMT solvers

- Satisfiability Modulo Theories: Akin to a SAT solver with support for additional theories: algebraic data types, integer arithmetic, real arithmetic, bitvectors, sets, etc.
- Can choose from Z3, CVC4, Yices, Princess, and others.
- In practice, cannot just translate from the host language into SMT because of quantifiers, recursive functions, polymorphism, etc.
- Lots of work, difficult to get right (ie. sound and complete).

Solving constraints with Inox[2]

1. Solver for higher-order functional programs which provides first-class support for features such as:
    1. Recursive and first-class functions
    2. ADTs, integers, bitvectors, strings, set-multiset-map abstractions
    3. Quantifiers
    4. ADT invariants
2. Implements a very involved *unfolding strategy* to deal with all of the above [1, 2, 3]
3. Powers Stainless, a verification system for Scala[1]

[1] https://github.com/epfl-lara/stainless
[2] https://github.com/epfl-lara/inox

Write your own language with refinement types

Demo

In the wild

LiquidHaskell

- Modern incarnation of refinement types for Haskell, ie. *Liquid types* [4]
- Refinement are quantifier-free predicates drawn from a decidable logic. [4]
- Type refinement are specified as comments in the source code.

LiquidHaskell

Demo

F*

- General-purpose dialect of ML with effects aimed at program verification.
- Dependently-typed language with refinements, type checking done via an SMT solver.
- Can be extracted to efficient OCaml, F, or C code.
- Initially developed at Microsoft Research.

F*

- Part of *Project Everest*, an in-progress verified implementation of HTTPS, TLS, X.509, and cryptographic algorithms.[3]

[3] https://project-everest.github.io

Scala 3 (one day?)

- Ongoing effort by Georg Schmid to add refinement types to Dotty/Scala 3[5]

Acknowledgement

Many thanks to Georg Schmid for his insights and for taking time to answer my questions.

Go check out his work! [5]

Thanks!

References I

[1] P. Suter, A. S. Köksal, and V. Kuncak, “Satisfiability modulo recursive programs,” in *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, Springer-Verlag, 2011.

[2] N. Voirol and V. Kuncak, “Automating verification of functional programs with quantified invariants,” p. 17, 2016.

[3] N. Voirol and V. Kuncak, “On satisfiability for quantified formulas in instantiation-based procedures,” 2016.

[4] R. Jhala, “Refinement types for haskell,” in *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification*, PLPV '14, pp. 27–27, ACM, 2014.

References II

[5] G. S. Schmid and V. Kuncak, "Smt-based checking of predicate-qualified types for scala," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pp. 31–40, ACM, 2016.