Refinement Types:
A Tutorial
Other titles in Foundations and Trends® in Programming Languages

*Shape Analysis*
Bor-Yuh Evan Chang, Cezara Drăgoi, Roman Manevich, Noam Rinetzky and Xavier Rival

*Progress of Concurrent Objects*
Hongjin Liang and Xinyu Feng
ISBN: 978-1-68083-672-1

*QED at Large: A Survey of Engineering of Formally Verified Software*
Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric and Zachary Tatlock

*Reconciling Abstraction with High Performance: A MetaOCaml approach*
Oleg Kiselyov
Refinement Types: A Tutorial

Ranjit Jhala
University of California, San Diego
USA
jhala@cs.ucsd.edu

Niki Vazou
IMDEA Software Institute, Madrid
Spain
niki.vazou@imdea.org

Full text available at: http://dx.doi.org/10.1561/2500000032
Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract Interpretation
- Compilation and Interpretation Techniques
- Domain Specific Languages
- Formal Semantics, including Lambda Calculi, Process Calculi, and Process Algebra
- Language Paradigms
- Mechanical Proof Checking
- Memory Management
- Partial Evaluation
- Program Logic
- Programming Language Implementation
- Programming Language Security
- Programming Languages for Concurrency
- Programming Languages for Parallelism
- Program Synthesis
- Program Transformations and Optimizations
- Program Verification
- Runtime Techniques for Programming Languages
- Software Model Checking
- Static and Dynamic Program Analysis
- Type Theory and Type Systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2021, Volume 6, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.
Dedicated to Tom Henzinger, on the occasion of his 60th birthday.
# Contents

1 Introduction ................................................. 2
   1.1 A Brief History ........................................ 4
   1.2 Goals & Outline ....................................... 5

2 Refinement Logic ............................................. 8
   2.1 Syntax .................................................. 8
   2.2 Semantics .............................................. 10
   2.3 Decidability ............................................ 11

3 The Simply Typed $\lambda$-calculus ......................... 12
   3.1 Examples .............................................. 12
   3.2 Types and Terms ...................................... 14
   3.3 Declarative Typing ................................... 15
   3.4 Verification Conditions ............................... 22
   3.5 Discussion ............................................ 26

4 Branches and Recursion ..................................... 29
   4.1 Examples .............................................. 29
   4.2 Types and Terms ...................................... 31
   4.3 Declarative Typing ................................... 31
   4.4 Verification Conditions ............................... 37
   4.5 Discussion ............................................ 38
# Refinement Inference

5.1 Examples .................................................. 41
5.2 Types and Terms ........................................ 45
5.3 Declarative Typing ....................................... 45
5.4 Verification Conditions ................................. 47
5.5 Solving Horn Constraints ............................... 50
5.6 Discussion .................................................. 52

# Type Polymorphism

6.1 Examples .................................................. 56
6.2 Types and Terms ........................................ 58
6.3 Declarative Typing ....................................... 59
6.4 Verification Conditions ................................. 63
6.5 Discussion .................................................. 65

# Data Types

7.1 Examples .................................................. 67
7.2 Types and Terms ........................................ 71
7.3 Declarative Typing ....................................... 73
7.4 Verification Conditions ................................. 80
7.5 Discussion .................................................. 84

# Refinement Polymorphism

8.1 Examples .................................................. 86
8.2 Types and Terms ........................................ 91
8.3 Declarative Typing ....................................... 93
8.4 Verification Conditions ................................. 99
8.5 Discussion .................................................. 101

# Termination

9.1 Examples .................................................. 103
9.2 Types and Terms ........................................ 108
9.3 Declarative Typing ....................................... 109
9.4 Verification Conditions ................................. 114
9.5 Discussion .................................................. 115
10 Programs as Proofs
  10.1 Examples ........................................ 118
  10.2 Types and Terms .................................. 125
  10.3 Declarative Checking ............................... 128
  10.4 Verification Conditions ............................ 129
  10.5 Discussion ......................................... 130

11 Related Work ........................................... 133
  11.1 Program Logic based Verifiers ....................... 133
  11.2 Refinement Type based Verifiers ..................... 134
  11.3 Soundness of Refinement Types ...................... 136

12 Conclusion .............................................. 138
  12.1 The Good: Types Enable Compositional Reasoning .... 138
  12.2 The Bad: Reasoning about State ...................... 140
  12.3 The Ugly: Explaining Verification Failures ........... 142

References ................................................ 145
Refinement Types: A Tutorial

Ranjit Jhala\textsuperscript{1} and Niki Vazou\textsuperscript{2}

\textsuperscript{1}University of California, San Diego, USA; jhala@cs.ucsd.edu
\textsuperscript{2}IMDEA Software Institute, Madrid, Spain; niki.vazou@imdea.org

ABSTRACT

Refinement types enrich a language’s type system with logical predicates that circumscribe the set of values described by the type. These refinement predicates provide software developers a tunable knob with which to inform the type system about what invariants and correctness properties should be checked on their code, and give the type checker a way to enforce those properties at compile time. In this article, we distill the ideas developed in the substantial literature on refinement types into a unified tutorial that explains the key ingredients of modern refinement type systems. In particular, we show how to implement a refinement type checker via a progression of languages that incrementally add features to the language or type system.
The type systems of modern languages like C#, Haskell, Java, Ocaml, Rust, and Scala are the most widely used method for establishing guarantees about the correct behavior of software. In essence, types allow the programmer to describe legal sets of values for various operations, thereby eliminating, at compile-time, the possibility of a large swathe of unexpected and undesirable run-time errors. Unfortunately, well-typed programs do go wrong.

1. **Divisions by zero** The fact that a divisor is an int does not preclude the possibility of a run-time divide-by-zero, or that a given arithmetic operation will over- or under-flow;

2. **Buffer overflows** The fact that an array or string index is an int does not eliminate the possibility of a segmentation fault, or worse, leaking data from an out-of-bounds access;

3. **Mismatched dimensions** Moving up a level, the fact that a product operator is given two matrix values does not prevent errors arising from the matrices having incompatible dimensions;

4. **Logic bugs** Classical type systems can ensure that each date structure contains suitable (e.g. int valued) fields holding the day,
month and year, but cannot guarantee that the day is valid for the
given month and year;

5. **Correctness errors** Finally, at the extreme end, a type system
can ensure that a sorting routine produces a list, and that a
compilation routine produces a sequence of machine instructions,
but cannot guarantee that the list was, in fact, an ordered per-
mutation of the input, or that the machine instructions faithfully
implemented the program source.

**Refining Types with Predicates** Refinement types allow us to enrich
a language’s type system with *predicates* that circumscribe the set of
values described by the type. For example, while an `int` can be any
integer value, we can write the refined type

```plaintext
type nat = int[v|0 <= v]
```

that describes only non-negative integers. By combining types and
predicates the programmer can write precise *contracts* that describe
legal inputs and outputs of functions. For example, the author of an
array library could specify that

```plaintext
val size : x: array(a) ⇒ nat[v|v = length(x)]
val get : x: array(a) ⇒ nat[v|v < length(x)] ⇒ a
```

which say that (1) a call `size(arr)` ensures the returned integer equals
to the number of elements in `arr`, and (2) the call `get(arr, i)` requires
the index `i` be within the bounds of `arr`. Given these specifications,
the refinement type checker can guarantee, at *compile time*, that all
operations respect their contracts, to ensure, e.g. that all array accesses
are safe at run-time.

**Language-Integrated Verification** Refinements provide a tunable
knob whereby developers can inform the type system about what invari-
ants and correctness properties they care about, *i.e.* are important for
the particular domain of their code. They could begin with basic safety
requirements, *e.g.* to eliminate divisions by zero and buffer overflows,
or ensure they don’t attempt to access values from an empty stack or
collection, and then, incrementally, dial the specifications up to include,
*e.g.* invariants about custom data types like dates, or ordered heaps, and,
if they desire, ultimately go all the way to specifying and verifying the correctness of various routines at compile-time. Crucially, (refinement) types eliminate the barrier between implementation and proof, by enabling verification within the same language, library and tool ecosystem. This tight integration is essential to create a virtuous cycle of feedback across the phases. The implementation dictates what properties are important, and provides hints on how to do the verification. Dually, the verification provides guidance on how the code can be restructured, \textit{e.g.} to make the abstractions and invariants explicit enough to enable formal proof.

1.1 A Brief History

Refinement types can be thought of as a type-based formulation of assertions from classical program logic (Turing, 1949; Floyd, 1967; Hoare, 1969). The idea of refining types with logical constraints goes back at least to Cartwright, 1976 who described a means of refining Lisp datatypes with constraints to aid in program verification. The ADA programming language has a notion of range types which allow the to define contiguous subsets of integers (Dewar \textit{et al.}, 1980). Nordstrom and Petersson, 1983 and Constable, 1983 introduced the notion of logical-refinements-as-subsets of values, and Constable, 1986 turned this notion into a pillar of the Nuprl proof assistant.

Freeman and Pfenning, 1991a introduced the name “refinement types” in a paper that describes a syntactic mechanism to define subsets of algebraic data\textsuperscript{1}. Inspired by the early work on Nuprl, the PVS proof assistant embraced the idea of types as subsets, and Rushby \textit{et al.}, 1998 introduced the notion of predicate subtyping which forms the basis of the subtyping relation that remains the workhorse of modern refinement type systems. Zenger, 1997 and Xi and Pfenning, 1998 describe a means of indexing types with (symbolic) integers after which constraints can be used to specify function contracts that can be verified by linear programming, to, \textit{e.g.} perform array bounds or list or matrix dimension checking at compile time, and Dunfield, 2007 shows how to combine

\textsuperscript{1}See Michael Greenberg’s post “A refinement type by any other name” for a more detailed discussion on the history of refinement types
indices with datasort refinements to facilitate the verification of data structure invariants.

The Sage system (Gronski et al., 2006) described how refinement like specifications could be verified in a hybrid manner: partly at compile time using SMT solvers, and partly at run-time via dynamic contract checks (Flanagan, 2006). Several groups picked up the gauntlet of moving all the checks to compile time, leading to the F7 (Bengtson et al., 2011) and then F* (Swamy et al., 2011) dialects of ML which has been used to formally verify the implementation of cryptographic routines used in widely used web-browsers (Zinzindohoué et al., 2017). Rondon et al., 2008 introduced the notion of liquid types which make refinements easier to use by delegating the task of synthesizing refinements to abstract interpretation.

The last decade has seen refinements spread over to languages outside the ML family. Rondon et al., 2010 and Chugh et al., 2012 show how to verify C and JavaScript programs by refining a low-level language of locations (Smith et al., 2000). Kent et al., 2016 show how refinements can be integrated within Racket’s occurrence based type system (Tobin-Hochstadt and Felleisen, 2008). Kazerounian et al., 2017 integrate refinements in Ruby’s type system using just-in-time type checking. Finally, Hamza et al., 2019 present a refinement-type based verifier for higher-order Scala programs.

1.2 Goals & Outline

Refinement types can be the vector that brings formal verification into mainstream software development. This happy outcome hinges upon the design and implementation of refinement type systems that can be retrofitted to existing languages, or co-designed with new ones. Our primary goal is to catalyze the development of such systems by distilling the ideas developed in the sprawling literature on the topic into a coherent and unified tutorial that explains the key ingredients of modern refinement type systems, by showing how to implement a refinement type checker.

**Background** We have tried to make this article as self-contained as possible. However, some familiarity with propositional logic and the
simply typed lambda calculus will be helpful.

A Nanopass Approach  Inspired by the nanopass framework for teaching compilation pioneered by Sarkar et al., 2004, we will show how to implement refinement types via a progression of languages that incrementally add features to the language or type system.

- \( \lambda_\phi \) (§ 3): We start with the simply typed \( \lambda \)-calculus, which will illustrate the foundations, namely, refinements, functions, and function application;

- \( \lambda_\beta \) (§ 4): Next, we will add branch conditions, and show how refinement type checkers do *path-sensitive* reasoning;

- \( \lambda_\kappa \) (§ 5): Types are palatable when we have to write down only the interesting ones: hence, next, we will see how to automatically *infer* the refinements to make using refinements pleasant;

- \( \lambda_\alpha \) (§ 6): After adding inference to our arsenal, we will be able to add type polymorphism which, will unlock various forms of *context-sensitive* reasoning;

- \( \lambda_\delta \) (§ 7): Once we have polymorphic types, we can add polymorphic *data types* like lists and trees, and see how to specify and verify properties of those structures;

- \( \lambda_\rho \) (§ 8): Type polymorphism allows us to reuse functions and data with different kinds of values. We will see why we often need to reuse functions and data across different kinds of invariants, and to support this, we will develop a form of *refinement polymorphism*;

- \( \lambda_\tau \) (§ 9): All of the above methods allow us to verify safety properties, *i.e.* assertions about values of code. Next, we will see how refinements let us verify *termination*;

- \( \lambda_\pi \) (§ 10): Finally, we will see how to write propositions over arbitrary user defined functions and write proofs of those propositions as well-typed programs, effectively converting the host language into a theorem prover.

Full text available at: http://dx.doi.org/10.1561/2500000032
1.2. Goals & Outline

Dependencies The ideal reader would, of course, devote several hours of thoughtful contemplation to each of the eight sub-languages. However, life is short, and you may be interested in particular aspects of refinement typing. If so, we suggest reading the chapters in the following order, summarized in Fig. 1.1

- § 3 and § 4 are essential, as they focus on the basics of refinement types and path-sensitive reasoning;
- § 5, § 6 and § 8 explain how to support polymorphism via refinement inference;
- § 7 explains how refinements allow reasoning about invariants of algebraic data types;
- § 9 and § 10 will be of interest to readers who wish to learn how to scale refinements up to proofs.

Implementation This article is accompanied by an implementation

https://github.com/ranjitjhala/sprite-lang

The README that accompanies the code has directions on how to build, modify and execute the sequence of type checkers that we will develop over the rest of this article. We welcome readers who like to get their hands dirty to clone the repository and follow along with the code.

And now, let’s begin!


Chugh, R., D. Herman, and R. Jhala. (2012). “Dependent Types for JavaScript”. In: OOPSLA.


Damas, L. and R. Milner. (1982b). “Principal Type-Schemes for Functional Programs”. In: POPL.


Pierce, B. C. and D. N. Turner. (1998). “Local Type Inference”. In: POPL.


Turing, A. M. (1936). “On computable numbers, with an application to the Entscheidungsproblem”. In: *LMS*.


Vekris, P., B. Cosman, and R. Jhala. (2016). “Refinement types for TypeScript”. In: PLDI.

Wadler, P. (2015). “Propositions As Types”. In: Communications of the ACM.


Xi, H. (2001). “Dependent Types for Program Termination Verification”. In: LICS.


