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
ISBN: 978-1-68083-732-2

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
ISBN: 978-1-68083-594-6

Reconciling Abstraction with High Performance: A MetaOCaml approach
Oleg Kiselyov
ISBN: 978-1-68083-436-9

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

the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

R. Jhala and N. Vazou. *Refinement Types: A Tutorial*. Foundations and Trends® in Programming Languages, vol. 6, no. 3-4, pp. 159–317, 2021.

ISBN: 978-1-68083-885-5

Foundations and Trends® in Programming Languages

Volume 6, Issue 3-4, 2021

Editorial Board

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

Refinement Types: A Tutorial

Ranjit Jhala[1] and Niki Vazou[2]

[1] *University of California, San Diego, USA; jhala@cs.ucsd.edu*
[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.

Ranjit Jhala and Niki Vazou (2021), "Refinement Types: A Tutorial", Foundations and Trends® in Programming Languages: Vol. 6, No. 3-4, pp 159–317. DOI: 10.1561/2500000032.

1

Introduction

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 permutation 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

```
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

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

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 invariants 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, *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 *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[1]. Inspired by the early work on Nuprl, the PVS proof assistant embraced the idea of types as subsets, and Rushby *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, *e.g.* perform array bounds or list or matrix dimension checking at compile time, and Dunfield, 2007 shows how to combine

[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.

- λ_ϕ (§ 3): We start with the simply typed λ-calculus, which will illustrate the foundations, namely, refinements, functions, and function *application*;
- λ_β (§ 4): Next, we will add branch conditions, and show how refinement type checkers do *path-sensitive* reasoning;
- λ_κ (§ 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;
- λ_α (§ 6): After adding inference to our arsenal, we will be able to add type polymorphism which, will unlock various forms of *context-sensitive* reasoning;
- λ_δ (§ 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;
- λ_ρ (§ 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*;
- λ_τ (§ 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*;
- λ_π (§ 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.

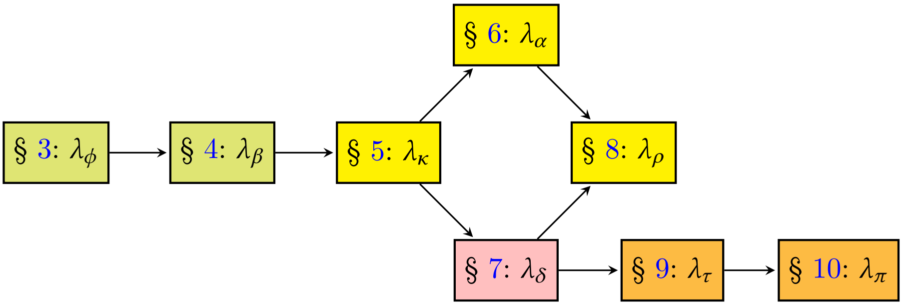

Figure 1.1: Chapter dependencies

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!

References

Amin, N., K. R. M. Leino, and T. Rompf. (2014). "Computing with an SMT Solver". In: *Tests and Proofs*.

Andrews, P. B. (2002). *An Introduction to Mathematical Logic and Type Theory*. Springer. URL: https://www.springer.com/gp/book/9781402007637.

Astrauskas, V., P. Müller, F. Poli, and A. J. Summers. (2019). "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. URL: https://doi.org/10.1145/3360573.

Baader, F. and T. Nipkow. (1998). *Term Rewriting and All That*. USA: Cambridge University Press.

Bakst, A. and R. Jhala. (2016). "Predicate Abstraction for Linked Data Structures". In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. *Lecture Notes in Computer Science*. Springer. 65–84. DOI: 10.1007/978-3-662-49122-5_3.

Barnett, M., M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. (2011). "Specification and Verification: The Spec# Experience". In: *Communications of the ACM*. URL: https://doi.org/10.1145/1953122.1953145.

Barrett, C., A. Stump, and C. Tinelli. (2010). "The SMT-LIB Standard: Version 2.0". In: *SMT*.

Barthe, G., M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. (2004). "Type-based termination of recursive definitions". In: *MSCS*.

Belo, J. F., M. Greenberg, A. Igarashi, and B. C. Pierce. (2011). "Polymorphic Contracts". In: *European Symposium on Programming (ESOP)*. URL: https://doi.org/10.1007/978-3-642-19718-5_2.

Bengtson, J., K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. (2011). "Refinement types for secure implementations". *ACM TOPLAS*.

Bertot, Y. and P. Castéran. (2004). *Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag.

Bierman, G., A. Gordon, C. Hriţcu, and D. Langworthy. (2010). "Semantic Subtyping with an SMT Solver". In: *International Conference on Functional Programming (ICFP)*. URL: https://doi.org/10.1145/1863543.1863560.

Lehmann, N., R. Kunkel, D. Stefan, and R. Jhala. (2020). "The Binah Web Framework".

Bird, R. S. (1989). "Algebraic Identities for Program Calculation". In: *The Computer Journal*.

Bjørner, N., A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. (2015). "Horn Clause Solvers for Program Verification". In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte. Vol. 9300. *Lecture Notes in Computer Science*. Springer. 24–51. DOI: 10.1007/978-3-319-23534-9_2.

Canning, P., W. Cook, W. Hill, W. Olthoff, and J. Mitchell. (1989). "Abstract F-Bounded Polymorphism for Object-Oriented Programming". In: DOI: 10.1145/99370.99392.

Cartwright, R. (1976). "User-Defined Data Types as an Aid to Verifying LISP Programs". In: *Third International Colloquium on Automata, Languages and Programming, University of Edinburgh, UK, July 20-23, 1976*. Ed. by S. Michaelson and R. Milner. Edinburgh University Press. 228–256.

Chugh, R., D. Herman, and R. Jhala. (2012). "Dependent Types for JavaScript". In: *OOPLSA*.

Clarke, D. G., J. Noble, and J. M. Potter. (2001). "Simple Ownership Types for Object Containment". In: *ECOOP 01: European Conference on Object Oriented Programming*. 53–76.

Cok, D. R. (2014). "OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse". In: *Electronic Proceedings in Theoretical Computer Science*. URL: http://dx.doi.org/10.4204/EPTCS.149.8.

"Collatz Conjecture". (2021).

Constable, R. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall.

Constable, R. L. (1983). "Mathematics as Programming". In: *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings*. Ed. by E. M. Clarke and D. Kozen. Vol. 164. *Lecture Notes in Computer Science*. Springer. 116–128. DOI: 10.1007/3-540-12896-4_359.

Cook, B., A. Podelski, and A. Rybalchenko. (2011). "Proving program termination". *Commun. ACM*.

Coquand, T. and G. Huet. (1985). "Constructions: A higher order proof system for mechanizing mathematics". In: *European Conference on Computer Algebra (EUROCAL)*. URL: https://doi.org/10.1007/3-540-15983-5_13.

Coquand, T. and G. Huet. (1988). "The Calculus of Constructions". In: *Information and Computation*. URL: https://doi.org/10.1016/0890-5401(88)90005-3.

Cosman, B. and R. Jhala. (2017). "Local refinement typing". *PACMPL*. 1(ICFP): 26:1–26:27. DOI: 10.1145/3110270.

Damas, L. and R. Milner. (1982a). "Principal Type-Schemes for Functional Programs." In: *POPL*.

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

Detlefs, D., G. Nelson, and J. B. Saxe. (2005). "Simplify: a theorem prover for program checking". *J. ACM*. 52(3): 365–473. DOI: 10.1145/1066100.1066102.

Dewar, R. B. K., G. A. Fisher, E. Schonberg, R. Froehlich, S. Bryant, C. F. Goss, and M. Burke. (1980). "The NYU Ada Translator and Interpreter". *SIGPLAN Not.* 15(11): 194–201. DOI: 10.1145/947783.948659.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice-Hall.

Dunfield, J. (2007). "A Unified System of Type Refinements". *PhD thesis.* Pittsburgh, PA, USA: Carnegie Mellon University.

Dunfield, J. (2017). "Extensible Datasort Refinements". In: *European Symposium on Programming (ESOP).* URL: https://doi.org/10.1007/978-3-662-54434-1_18.

Dunfield, J. and N. Krishnaswami. (2020). "Bidirectional Typing". In:

Dunfield, J. and N. R. Krishnaswami. (2013). "Complete and easy bidirectional typechecking for higher-rank polymorphism". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013.* Ed. by G. Morrisett and T. Uustalu. ACM. 429–442. DOI: 10.1145/2500365.2500582.

Filliâtre, J. (1998). "Proof of Imperative Programs in Type Theory". In: *Types for Proofs and Programs, International Workshop TYPES '98, Kloster Irsee, Germany, March 27-31, 1998, Selected Papers.* Ed. by T. Altenkirch, W. Naraschewski, and B. Reus. Vol. 1657. *Lecture Notes in Computer Science.* Springer. 78–92. DOI: 10.1007/3-540-48167-2_6.

Flanagan, C. (2006). "Hybrid Type Checking". In: *POPL.*

Flanagan, C. and K. Leino. (2001). "Houdini, an Annotation Assistant for ESC/Java". URL: citeseer.ist.psu.edu/flanagan00houdini.html.

Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. (1993). "The Essence of Compiling with Continuations." In: *PLDI.*

Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. (2002). "Extended Static Checking for Java". In: *Programming Language Design and Implementation (PLDI).* URL: https://doi.org/10.1145/512529.512558.

Floyd, R. (1967). "Assigning meanings to programs". In: *Mathematical Aspects of Computer Science.*

Freeman, T. and F. Pfenning. (1991a). "Refinement Types for ML". In: *PLDI.*

Freeman, T. and F. Pfenning. (1991b). "Refinement Types for ML". In: *Programming Language Design and Implementation (PLDI)*. URL: https://doi.org/10.1145/113445.113468.

Giesl, J., M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. (2011). "Automated termination proofs for Haskell by term rewriting". In: *TPLS*.

Gopan, D., T. W. Reps, and S. Sagiv. (2005). "A framework for numeric analysis of array operations." In: *POPL*. 338–350.

Gordon, A. D. and C. Fournet. (2010). "Principles and Applications of Refinement Types". In: *Logics and Languages for Reliability and Security*. IOS Press. URL: https://doi.org/10.3233/978-1-60750-100-8-73.

Gordon, C. S., M. D. Ernst, D. Grossman, and M. J. Parkinson. (2017). "Verifying Invariants of Lock-Free Data Structures with Rely-Guarantee and Refinement Types". *ACM Trans. Program. Lang. Syst.* 39(3): 11:1–11:54. DOI: 10.1145/3064850.

Graf, S. and H. Saidi. (1997). "Construction of abstract state graphs with PVS". In: *CAV. LNCS 1254*. Springer. 72–83.

Gronski, J., K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. (2006). "Sage: Hybrid checking for flexible specifications". In: *Scheme and Functional Programming Workshop*. 93–104.

Gulwani, S., O. Polozov, and R. Singh. (2017). "Program Synthesis". *Found. Trends Program. Lang.* 4(1-2): 1–119. DOI: 10.1561/2500000010.

Gurfinkel, A. and N. Bjørner. (2019). "The Science, Art, and Magic of Constrained Horn Clauses". In: *21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2019, Timisoara, Romania, September 4-7, 2019*. IEEE. 6–10. DOI: 10.1109/SYNASC49474.2019.00010.

Hallahan, W. T., A. Xue, M. T. Bland, R. Jhala, and R. Piskac. (2019). "Lazy counterfactual symbolic execution". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM. 411–424. DOI: 10.1145/3314221.3314618.

Hamza, J., N. Voirol, and V. Kuncak. (2019). "System FR: formalized foundations for the stainless verifier". *Proc. ACM Program. Lang.* 3(OOPSLA): 166:1–166:30. DOI: 10.1145/3360592.

Handley, M. A. T., N. Vazou, and G. Hutton. (2019). "Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/3371092.

Hindley, R. (1969). "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society*.

Hoare, C. A. R. (1971). "Procedures and parameters: An axiomatic approach". In: *Symposium on Semantics of Algorithmic Languages*.

Hoare, C. (1969). "An Axiomatic Basis for Computer Programming". *Communications of the ACM*. 12: 576–580.

Hoder, K. and N. Bjørner. (2012). "Generalized Property Directed Reachability". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. *Lecture Notes in Computer Science*. Springer. 157–171. DOI: 10.1007/978-3-642-31612-8_13.

Hojjat, H. and P. Rümmer. (2018). "The ELDARICA Horn Solver". In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by N. Bjørner and A. Gurfinkel. IEEE. 1–7. DOI: 10.23919/FMCAD.2018.8603013.

Howard, W. A. (1980). "The formulae-as-types notion of construction". In: *Essays on Combinatory Logic, Lambda Calculus and Formalism*.

Hughes, J., L. Pareto, and A. Sabry. (1996). "Proving the Correctness of Reactive Systems Using Sized Types". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '96*. St. Petersburg Beach, Florida, USA: Association for Computing Machinery. 410–423. DOI: 10.1145/237721.240882.

Ishtiaq, S. S. and P. W. O'Hearn. (2001). "BI as an Assertion Language for Mutable Data Structures." In: *POPL*. 14–26.

Jhala, R. and K. McMillan. (2006). “A Practical and Complete Approach to Predicate Refinement”. In: *TACAS 06. LNCS 2987*. Springer-Verlag. 298–312.

Jhala, R., A. Podelski, and A. Rybalchenko. (2018). “Predicate Abstraction for Program Verification”. In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. Cham: Springer International Publishing. 447–491. DOI: 10.1007/978-3-319-10575-8_15.

Jones, N. D. and N. Bohr. (2004). “Termination Analysis of the Untyped lamba-Calculus”. In: *RTA*.

Jung, R., R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. (2018). “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. *J. Funct. Program.* 28: e20. DOI: 10.1017/S0956796818000151.

Kazerounian, M., N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak. (2017). “Refinement Types for Ruby”. *CoRR*. abs/1711.09281. arXiv: 1711.09281. URL: http://arxiv.org/abs/1711.09281.

Kent, A. M., D. Kempe, and S. Tobin-Hochstadt. (2016). “Occurrence typing modulo theories”. In: *PLDI*.

Kloos, J., R. Majumdar, and V. Vafeiadis. (2015). “Asynchronous Liquid Separation Types”. In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Ed. by J. T. Boyland. Vol. 37. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 396–420. DOI: 10.4230/LIPIcs.ECOOP.2015.396.

Knowles, K. W. and C. Flanagan. (2009). “Compositional and decidable checking for dependent contract types”. In: *PLPV*.

Knowles, K. W. and C. Flanagan. (2010). “Hybrid type checking”. In: *TOPLAS*.

Komondoor, R., G. Ramalingam, S. Chandra, and J. Field. (2005). “Dependent Types for Program Understanding.” In: *TACAS*. 157–173.

Komuravelli, A., A. Gurfinkel, and S. Chaki. (2016). “SMT-based model checking for recursive programs”. *Formal Methods Syst. Des.* 48(3): 175–205. DOI: 10.1007/s10703-016-0249-4.

Kroening, D. and O. Strichman. (2008). *Decision Procedures: An Algorithmic Point of View*. Springer.

Lahiri, S. K., S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. (2009). "Intra-module Inference". In: *Computer Aided Verification*. Ed. by A. Bouajjani and O. Maler. Berlin, Heidelberg: Springer Berlin Heidelberg. 493–508.

Lehmann, N., R. Kunkel, J. Brown, J. Yang, N. Vazou, N. Polikarpova, D. Stefan, and R. Jhala. (2021). "STORM: Refinement Types for Secure Web Applications". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association. 441–459. URL: https://www.usenix.org/conference/osdi21/presentation/lehmann.

Leino, K. R. M. and G. Nelson. (1998). "An Extended Static Checker for Modula-3". In: *Compiler Construction*.

Leino, K. R. M. and C. Pit-Claudel. (2016a). "Trigger selection strategies to stabilize program verifiers". In: *CAV*.

Leino, K. R. M. and N. Polikarpova. (2016). "Verified Calculations". In: *VSTTE*.

Leino, K. R. M. (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. URL: https://doi.org/10.1007/978-3-642-17511-4_20.

Leino, R. and C. Pit-Claudel. (2016b). "Trigger Selection Strategies to Stabilize Program Verifiers". In: *Computer Aided Verification (CAV)*. URL: https://doi.org/10.1007/978-3-319-41528-4_20.

Ley-Wild, R. and A. Nanevski. (2013). "Subjective Auxiliary State for Coarse-Grained Concurrency". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '13*. Rome, Italy: Association for Computing Machinery. 561–574. DOI: 10.1145/2429069.2429134.

Li, Y., T. Tan, A. Møller, and Y. Smaragdakis. (2020). "A Principled Approach to Selective Context Sensitivity for Pointer Analysis". *ACM Trans. Program. Lang. Syst.* 42(2). DOI: 10.1145/3381915.

Liu, Y., J. Parker, P. Redmond, L. Kuper, M. Hicks, and N. Vazou. (2020a). "Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell". In: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. URL: https://doi.org/10.1145/3428284.

Liu, Y., J. Parker, P. Redmond, L. Kuper, M. Hicks, and N. Vazou. (2020b). "Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell". *Proc. ACM Program. Lang.* 3(OOPSLA).

Lovas, W. and F. Pfenning. (2010). "Refinement Types for Logical Frameworks and Their Interpretation as Proof Irrelevance". In: *Logical Methods in Computer Science*. URL: http://dx.doi.org/10.2168/LMCS-6(4:5)2010.

Maillard, K., D. Ahman, R. Atkey, G. Martínez, C. Hriţcu, E. Rivas, and É. Tanter. (2019). "Dijkstra Monads for All". In: *International Conference on Functional Programming (ICFP)*. URL: https://doi.org/10.1145/3341708.

Martínez, G., D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. (2019). "Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms". In: *European Symposium on Programming (ESOP)*. URL: https://doi.org/10.1007/978-3-030-17184-1%5C_2.

McCarthy, J. (1962). "Towards a Mathematical Science of Computation". In: *IFIP*.

Melliès, P.-A. and N. Zeilberger. (2015). "Functors Are Type Refinement Systems". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/2676726.2676970.

Might, M. (2007). "Logic-flow analysis of higher-order programs". In: *POPL*. 185–198.

Milner, R. (1978). "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences*. URL: https://doi.org/10.1016/0022-0000(78)90014-4.

Mu, S. C., H. S. Ko, and P. Jansson. (2009). "Algebra of Programming in Agda: Dependent Types for Relational Program Derivation". In: *J. Funct. Program.*

Nanevski, A., G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. (2008a). "Ynot: dependent types for imperative programs". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by J. Hook and P. Thiemann. ACM. 229–240. DOI: 10.1145/1411204.1411237.

Nanevski, A., J. G. Morrisett, and L. Birkedal. (2008b). "Hoare type theory, polymorphism and separation". *J. Funct. Program.* 18(5-6): 865–911. DOI: 10.1017/S0956796808006953.

Necula, G. C. (1997). "Proof carrying code". In: *POPL 97: Principles of Programming Languages*. ACM. 106–119.

Nelson, C. G. (1980). "Techniques for Program Verification". *PhD thesis*. Stanford University.

Nguyen, P. C., T. Gilray, S. Tobin-Hochstadt, and D. Van Horn. (2019). "Size-Change Termination as a Contract: Dynamically and Statically Enforcing Termination for Higher-Order Programs". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2019*. Phoenix, AZ, USA: Association for Computing Machinery. 845–859. DOI: 10.1145/3314221.3314643.

Nordstrom, B. and K. Petersson. (1983). "Types and Specifications". In: *IFIP*.

Ou, X., G. Tan, Y. Mandelbaum, and D. Walker. (2004). "Dynamic Typing with Dependent Types". In: *IFIP TCS*.

Parker, J., N. Vazou, and M. Hicks. (2019a). "LWeb: Information Flow Security for Multi-Tier Web Applications". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/3290388.

Parker, J., N. Vazou, and M. Hicks. (2019b). "LWeb: information flow security for multi-tier web applications". *Proc. ACM Program. Lang.* 3(POPL): 75:1–75:30. DOI: 10.1145/3290388.

Paulson, L. C., T. Nipkow, and M. Wenzel. (2019). "From LCF to Isabelle/HOL". In: *Formal Aspects of Computing*. URL: https://doi.org/10.1007/s00165-019-00492-1.

Peyton-Jones, S. L., D. Vytiniotis, S. Weirich, and G. Washburn. (2006). "Simple unification-based type inference for GADTs". In: *ICFP*.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

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

Pierce, B. (2003). "Types and Programming Languages: The Next Generation". In: *Logic in Computer Science*. IEEE Computer Society. 32. DOI: 10.1109/LICS.2003.1210042.

Plotkin, G. (1977). "LCF considered as a programming language". In: *Theoretical Computer Science*. URL: http://www.sciencedirect.com/science/article/pii/0304397577900445.

Podelski, A. and A. Rybalchenko. (2004). "Transition Invariants". In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science. LICS '04*. USA: IEEE Computer Society. 32–41.

Polikarpova, N., I. Kuraj, and A. Solar-Lezama. (2016). "Program synthesis from polymorphic refinement types". In: *PLDI*.

Polikarpova, N., D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama. (2020). "Liquid information flow control". *Proc. ACM Program. Lang.* 4(ICFP): 105:1–105:30. DOI: 10.1145/3408987.

Potanin, A., J. Östlund, Y. Zibin, and M. D. Ernst. (2013). "Immutability". In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by D. Clarke, J. Noble, and T. Wrigstad. Vol. 7850. *Lecture Notes in Computer Science*. Springer. 233–269. DOI: 10.1007/978-3-642-36946-9_9.

Protzenko, J., B. Beurdouche, D. Merigoux, and K. Bhargavan. (2019). "Formally Verified Cryptographic Web Applications in WebAssembly". In: *Security and Privacy (SP)*. URL: http://dx.doi.org/10.1109/SP.2019.00064.

Qiu, X., P. Garg, A. Stefanescu, and P. Madhusudan. (2013). "Natural proofs for structure, data, and separation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by H. Boehm and C. Flanagan. ACM. 231–242. DOI: 10.1145/2491956.2462169.

Reynolds, J. C. (2002). "Separation Logic: A Logic for Shared Mutable Data Structures." In: *LICS*. 55–74.

Rice, H. G. (1953). "Classes of Recursively Enumerable Sets and Their Decision Problems". *Transactions of the American Mathematical Society*. 74(2): 358–366. URL: http://www.jstor.org/stable/1990888.

Rondon, P., M. Kawaguchi, and R. Jhala. (2008). "Liquid Types". In: *PLDI*.

Rondon, P., M. Kawaguchi, and R. Jhala. (2010). "Low-Level Liquid Types". In: *POPL*.

Ruemmer, P. (2021). "Constrained Horn Clause Solver Competition".

Rushby, J., S. Owre, and N. Shankar. (1998). "Subtypes for Specifications: Predicate Subtyping in PVS". In:

Sagiv, S., T. W. Reps, and R. Wilhelm. (2002). "Parametric shape analysis via 3-valued logic." *ACM Trans. Program. Lang. Syst.* 24(3): 217–298.

Sarkar, D., O. Waddell, and R. K. Dybvig. (2004). "A Nanopass Infrastructure for Compiler Education". *SIGPLAN Not.* 39(9): 201–212. DOI: 10.1145/1016848.1016878.

Schrijvers, T., S. L. Peyton-Jones, M. Sulzmann, and D. Vytiniotis. (2009). "Complete and decidable type inference for GADTs". In: *ICFP*.

Sekiyama, T., Y. Nishida, and A. Igarashi. (2015). "Manifest Contracts for Datatypes". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by S. K. Rajamani and D. Walker. ACM. 195–207. DOI: 10.1145/2676726.2676996.

Sereni, D. and N. Jones. (2005). "Termination analysis of higher-order functional programs". In: *APLAS*.

Shivers, O. (1988). "Control-Flow Analysis in Scheme". In: *PLDI*.

Smith, F., D. Walker, and J. Morrisett. (2000). "Alias Types". In: *ESOP*.

Sozeau, M., S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. (2020). "Coq Coq Correct. Verification of Type Checking and Erasure for Coq, in Coq". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/3371076.

Sulzmann, M., M. Odersky, and M. Wehr. (1997). "Type Inference with Constrained Types". In: *FOOL*. URL: citeseer.ist.psu.edu/article/odersky99type.html.

Suter, P., A. S. Köksal, and V. Kuncak. (2011). "Satisfiability Modulo Recursive Programs". In: *SAS*.

Swamy, N., J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. (2011). "Secure distributed programming with value-dependent types". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by M. M. T. Chakravarty, Z. Hu, and O. Danvy. ACM. 266–278. DOI: 10.1145/2034773.2034811.

Swamy, N., C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. (2016). "Dependent Types and Multi-Monadic Effects in F*". In: *Principles of Programming Languages (POPL)*. URL: https://doi.org/10.1145/2837614.2837655.

Timany, A. and M. Sozeau. (2018). "Cumulative Inductive Types in Coq". In: *Formal Structures for Computation and Deduction (FSCD)*. URL: https://doi.org/10.4230/LIPIcs.FSCD.2018.29.

Tobin-Hochstadt, S. and M. Felleisen. (2008). "The design and implementation of typed scheme". In: *POPL*.

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

Turing, A. (1949). "Checking a Large Routine". In: *The Early British Computer Conferences*. Cambridge, MA, USA: MIT Press. 70–72.

Jhala, R. (2019). "Types vs. Floyd-Hoare".

Vazou, N., A. Bakst, and R. Jhala. (2015). "Bounded refinement types". In: *ICFP*.

Vazou, N., L. Lampropoulos, and J. Polakow. (2017). "A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq". In: *Haskell*.

Vazou, N., P. Rondon, and R. Jhala. (2013). "Abstract Refinement Types". In: *ESOP*.

Vazou, N., E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton-Jones. (2014a). "Refinement Types for Haskell". In: *ICFP*.

Vazou, N., E. L. Seidel, and R. Jhala. (2014b). "LiquidHaskell: Experience with Refinement Types in the Real World". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. Haskell '14*. Gothenburg, Sweden: Association for Computing Machinery. 39–51. DOI: 10.1145/2633357.2633366.

Vazou, N., A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. (2018). "Refinement reflection: complete verification with SMT". *Proc. ACM Program. Lang.* 2(POPL): 53:1–53:31. DOI: 10.1145/3158141.

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*.

Wadler, P. (1989). "Theorems for Free". In: *Functional Programming Languages and Computer Architecture (FPCA)*. URL: https://doi.org/10.1145/99370.99404.

Walker, D. and J. Morrisett. (2000). "Alias Types for Recursive Data Structures". In: *Types in Compilation (TIC)*.

Wenzel, M. (2016). "The Isabelle System Manual". URL: https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2016-1/doc/system.pdf.

Winant, T., D. Devriese, F. Piessens, and T. Schrijvers. (2014). "Partial Type Signatures for Haskell". In: *Practical Aspects of Declarative Languages (PADL)*. Ed. by M. Flatt and H.-F. Guo. Vol. 8324. *Lecture Notes in Computer Science*. Springer. 17–32. URL: /Research/papers/padl2014.pdf.

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

Xi, H. and F. Pfenning. (1998). "Eliminating Array Bound Checking Through Dependent Types". In: *PLDI*.

Zeilberger, N. (2016). "Principles of Refinement Types". In: *Oregon Programming Languages Summer School (OPLSS)*. URL: https://www.cs.bham.ac.uk/~zeilbern/oplss16/refinements-notes.pdf.

Zenger, C. (1997). "Indexed Types". *TCS*.

Zhu, H., S. Magill, and S. Jagannathan. (2018). "A Data-Driven CHC Solver". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2018*. Philadelphia, PA, USA: Association for Computing Machinery. 707–721. DOI: 10.1145/3192366.3192416.

Zinzindohoué, J. K., K. Bhargavan, J. Protzenko, and B. Beurdouche. (2017). "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM. 1789–1806. DOI: 10.1145/3133956.3134043.