DTP Studentship Proposal: Intensional Refinement Types

Steven Ramsay

Research area: static program analysis

Computer software is complex, and the result of our inability to properly manage that complexity is an unacceptable rate of defects. The traditional approach to reducing the incidence of software defects is a combination of testing and debugging. However, this approach is fundamentally limited. Not only is it infeasible to execute a program under all conditions, but merely exhausting all expected inputs is already prohibitively expensive.

Static program analysis is an alternative approach, based on mathematical techniques, that has started gaining traction in industry over the last decade. Static program analysis tools are now used by most major software companies to catch bugs that would be too difficult (or expensive) to detect by testing alone.

Where traditional testing relies on human effort and ingenuity, static program analysis is underpinned by formal logic and advanced algorithms for automated reasoning. However, developing algorithms to reason about programs effectively is notoriously difficult. In theory, most of the problems in this domain are undecidable; in practice, commercial software is big and reasoning about its behaviour holistically, and under all conditions, is extremely complex.

This combination of technical depth and realworld impact makes static program analysis a large and vibrant research area spanning programming languages, verification and mathematical logic. At POPL — the premier venue for research in programming languages and compilers — 8 out of the top 10 most cited publications are on the subject of static program analysis [1].

Limits of current theory and practice

To manage the enormous complexity, algorithms for static program analysis rely on carefully engineered mathematical abstractions of programs. Great advances have been made in our understanding of effective abstractions for hardware and low-level, imperative programming (e.g. C and C++ programs). However, major challenges are still to be overcome in the static analysis of functional programming languages (popular examples include Scala, Haskell, Clojure, OCaml, F#).

Functional programs underpin critical areas of financial services, energy and scientific computing, so the realisation of static analysis technologies is remarkably important. Moreover, there has been a marked increase in functional programming within languages that are not traditionally considered to be functional, largely thanks to the growth of multicore/GPGPU processing and cloud computing for data analysis.

One of the most successful approaches to the static analysis of functional programs are refinement type systems. Refinement type systems build abstractions of functional programs through the branch of mathematical logic called type theory. However, state of the art static analysis based on refinement type systems does not scale to the sizes of program regularly seen in industrial contexts. For example, LiquidHaskell [5], which is one of the best known systems in academia, requires programmers to provide extensive help to the tool by annotating their programs with hundreds of lines of logical formulae that describe the behaviour of functions — a task that is infeasible for most programmers.
Project proposal

The current generation of refinement type systems fail to scale to real-world software because they do not exploit the compositionality of functional programs — programs are constructed in a modular fashion and, according to best practice, components have relatively simple interfaces. Building on new results from the supervisor’s recent POPL paper [2], this project proposes to develop a new class of system, called intensional refinement type systems, that exploiting compositionality: investigating their foundations, algorithms for automated reasoning and practical tools for working programmers. The central goals are as follows.

1. **Foundations** To develop a new system of intensional refinement types, which exploit the compositionality of programs by restricting the language of type constraints at interfaces to promote automatic generalisation. To analyse the expressive power of the system compared to existing refinement type systems and to understand the algorithmic complexity of type inference.

2. **Algorithms** To develop algorithms for solving this new class of restricted constraint. Directed by case studies, to understand the classes of constraint that occur in practice and to develop efficient algorithms that exploit their features. To design approximate methods and heuristics to solve difficult patterns that occur frequently. To work with the supervisor’s collaborators at the University of Oxford to improve the state of the art of higher-order constraint solving.

3. **Tools** To support the industrial-strength, functional and object-oriented programming language F# and implement tools to make static analysis practical for working programmers. To extend the system from a pure fragment to the full, industrial programming language F#. To conduct a large case study on open-source software and to work with commercial developers to improve the practical applicability of the tools.

---


