Research Statement

Shachar Itzhaky

My research agenda is to design programming systems to support the tremendous effort of developing software today. I want to build on our growing understanding of automated logical reasoning and make it applicable to real-world systems, drawing on user insight through interaction to address complicated reasoning problems and scalability issues. Users should be able to harness as much automation as the system has to offer, without having to resort to completely manual methods in situations where the system fails to deliver: instead, the user provides more hints that guide the systems. This is largely done in two ways: (a) pointing out the general structure of the solution or a programming construct to use, such as recursion, memoization, etc., and (b) by decomposing a large problem into smaller, easier to handle ones. Human programmers usually develop good intuition concerning the latter, and such hints are not only invaluable for purposes of automation, but also serve as documentation for program understanding during the software life cycle. The hope is that such systems will improve software quality and reliability, while also increasing productivity and reducing costs.

My expertise draws from two primary areas: my doctoral work on the use of decidable logics to reason about complicated program invariants, and my later work on using synthesis, essentially involving undecidable problems but allowing a higher level of reasoning. A mixture of the two can be very appealing since we can identify interesting sub-problems in the overall process that can be formulated in a decidable logic, providing a more predictable behavior for automatic systems. For example, checking preconditions for simplifying expressions, such as the condition "$x * a < 0$", which can be simplified to "$x < 0$" if a is known to be positive. This precondition can be decided via a procedure for integer linear arithmetic. The message is, that an undecidable problem can become more friendly, if we introduce bits and pieces of decidable sub-problems to construct the solution from.

The vision of a programming system, as I see it, is of an environment that allows high-level manipulation of computer programs using mathematical principles and axioms, based on a theory of sound transformation. The underlying philosophy is that programs and program data are mathematical objects, hence our usual abstractions from the math world would apply to them. That said, computer programs are fundamentally different from mathematical equations due to the way we use programming languages as opposed to the language of mathematics. In math, the trivial parts are usually skimmed through using natural language, introducing formalism where it serves an acute purpose; in computer programming, everything must compile, so the finest detail has to be spelled out in a formal language. The latter is also true for machine-checked proofs, which are regarded to be a particular kind of computer programs. The consequence is that computer programs grow quite large, but they are full of mundane details that do not convey any insight about the system – most frequently, they obscure the high-level concepts that underlie it. For this reason, the traditional pencil-and-paper approach used by mathematicians is impractical. At the same time, while mathematics is extremely versatile, having many different fields and sub-fields which vary greatly, software reasoning revolves mostly around first- and second-order logic, some arithmetic, and very basic algebra. There are, of course, niche applications of numerical nature that require more background theory, but their treatment can often be localized to small parts of the code. This provides an opportunity for specialization of generic proof assistants toward their more extensive use in software development.

My main insight is that human intuition is a vital ingredient in the process, hence a lot of consideration has to be put into interactivity aspects. At the same time, the growing availability of computing power, as well as recent developments in constraint solving technology, provide great opportunities for automation. I intend to focus on software synthesis approaches, which have shown great promise in the last decade as they gradually scale up from code snippets of just a few instructions to real-world programs including ciphers, database transactions, and image processing kernels. These can leverage human intuition and creativity while reducing the volume of mundane, repetitive, and monotonous tasks usually associated with the programming process.

With these tools I have a vision to tackle a long-standing problem of programming by refinement. The concept of refinement was suggested by computer scientists of the 20th century, but was not brought to fruition and eventually went out of style, mostly due to the large amount of menial effort required by the methods of the time. According to the refinement discipline, software is structured by starting at a very high level of abstraction, where it is easy to convince oneself that desired functionality is specified correctly.

This layer is then refined to get it closer to an implementation by adding more details. By a process of gradual step-wise refinement, the programmer eventually arrives at a full implementation of the system. At each layer, a formal proof of correctness, relative to the layer above, is provided. In the early refinement systems, proof had to be supplied by the developer, which made the process very rigorous and demanding. Today, however, we are equipped with a rich toolbox of automated reasoning techniques that can not only find many of the proofs, but also suggest pieces and building blocks for the next layer of the refinement.

Refinement guides the development process by making the programmer's intuition explicit and anchoring it in the code. The artifact is thus not just the lowest layer, but the entire construction, which can greatly improve the understandability and maintainability of the software. Large systems obtained this way are easier to debug due to greater introspection capabilities, e.g. the ability to request running traces or logs at a high level of abstraction and have the data automatically collected from the lower levels. I envision this becoming the new "gold standard" for high quality software; whereas today the standard is to have design documents, a test suite, etc., the expectation may be, in the future, to have a refinement-style break down of the system into traceable layers with machine-verified proofs.

Past and Present Projects

EPR reasoning for linked data structures (Ph.D. thesis). During my doctoral studies, I was researching a way to automatically reason about pointer reachability properties in a complete way that always terminates, and either reports validation of the required property or provides the user with a counterexample that de-proves it. For this purpose, I managed to formulate the problem using effectively propositional logic, a decidable fragment of first-order logic, in a way that accommodates many natural properties of linked lists and several other linked data structures such as union-find (used, for example, in Kruskal's algorithm). As an extension and to improve usability, my collaborations led me to an integration of the method in Bradley's IC3 algorithm to facilitate automatic discovery of loop invariants in programs. This work has been used as a basis to follow-up work at the PL group at Tel Aviv University, and is used as the core in some ongoing projects.

Bellmania. As a demonstration of the efficacy of synthesis in non-trivial scenarios and as a response to a challenge introduced by Prof. Charles Leiserson at MIT, I designed an developed a formal framework for mechanized derivation of cache-oblivious implementations for sequential and parallel dynamic programming algorithms. Dynamic programming is pervasive in several applications of computer science and can greatly speed up otherwise intractable computations, but requires large amounts of memory, which puts stress on memory access bandwidths and creates performance bottlenecks. These problems can be alleviated by more effective utilization of the machine's fast memory caches, but doing so requires complex reordering of memory accesses to achieve desired spatial and temporal locality properties. By building up on recent developments in parallel high-performance computing, in particular recursive divide-and-conquer applied to dynamic programming, and by restating it as a refinement problem that starts from a naïve, inefficient loop implementation and targets a better, equivalent implementation, which is also cache-oblivious. I encoded the concepts underlying the divide-and-conquer technique as program transformation steps set up in an interactive environment where the user chooses which transformations to apply and the system continuously verifies semantic soundness of each step. This allowed me to produce verified implementations of several dynamic programming algorithms using a handful of transformation rules.

TransCal. In an effort to extend the efficacy of interactive derivational synthesis beyond the domain of dynamic programming, I am currently developed a calculus that can be used in the context of a "Synthesis Assistant", which I consider as the software development equivalent of proof assistants the likes of Coq and Isabelle/HOL. The calculus is based on a set of rewrite rules that induce a program equivalence graph, that is, a compact representation of many equivalent programs for functional program terms and sub-terms. With TransCal, the user is manipulating a functional program by exploring the space of equivalences and selecting a rewrite step to take next, as well as indicating particular sub-terms for generalization into new rewrite rules. The system compiles these generalized terms into subroutines; for example, if the user marks a sub-term as "$\mathit{fact}\ n$" and later rewrites the term as "$n * \mathit{fact}\ (n-1)$", then the extracted code would be

"$\mathit{fact}\ n =$ if $n > 0$ then $n * \mathit{fact}\ (n-1)$ else _". Filling in the placeholder "_" for the corner case involved solving a local synthesis problem, which would typically be small and therefore tractable.

Object Spreadsheets. This is a new programming model based on reactive programming and focusing on empowering a pervasive reactive environment – the spreadsheet. Although not originally thought of as a programming tool, spreadsheets have become considered as such as more programming-like functionalities are added to them. In my collaboration with the Software Design Group at MIT CSAIL, we sought to extend the data model provided by spreadsheets so it can be used in application development. Code requires structure, and traditional spreadsheets basically offer a two-dimensional array where semantics of cells are implicit, and any nontrivial data access require the use of functions that compute cell addresses. In Object Spreadsheets, we incorporated the notion of a schema, based on an entity-relationship model, similar to ones used in database system design and XML service oriented architectures. This structure admits hierarchy, which we identified as painfully missing from traditional spreadsheets while desired by most applications. We adapted the formula language to intrinsically support the new structure, such that referencing cells across the hierarchies and object references is natural, while preserving the functional, reactive nature of spreadsheet formulas, where change propagates through data dependencies. We evaluated our tool by implementing Web applications designed according to requirements from real-life scenarios were routine organizational tasks, such as scheduling a parent-teacher conference or cycling through group members to restock supplies, can benefit from collaborative automation. As a vision, by simplifying the developer interface and minimizing the amount of code that has to be written, Object Spreadsheets would enable end users to write applications that suit their needs with the same ease as filling a spreadsheet.

Future Research Plans

I plan to build on TransCal and continue to evolve it, incorporating more automation and scaling the mechanisms to support larger use cases. I intend to explore the application of inductive synthesis techniques — ones that observe the program's behavior on a set of constructed inputs — to draw conclusions about rewrite rules that can be used and guide the search process more efficiently. In the presence of non-terminating rewrite systems, for example, "$x = -(-x)$", the program equivalence graph cannot be saturated and success to find the desired solution depends very much on the ranking of rules and terms, such that more likely steps are explored first. Rewrites that grow the term arbitrarily are considered harmful and the system should avoid them if possible; however, these are occasionally required to reach the goal, so they cannot be deferred indefinitely.

A central issue to the applicability of a synthesis assistant to the software development process is the design of an effective user interface, in particular one that exhibits discoverability and, to the extent possible, isomorphism with users' perceived concepts and thought patterns. Contemporary proof assistants struggle to achieve adoption due to a steep learning curve and too little investment in user experience, especially for learners. Designing such an interface for a programming system would have to involve some form of user study; although for a development tool that typically requires its user to read some tutorial or manual to learn how to use it, the common model of studies involving 1–2 hour sessions with participants is impractical. A better environment for such evaluation would be a workshop or as an extra credit project in class.

The level of automation will distinguish these synthesis assistants from existing synthesis approaches based on proof assistants like Coq. On the other hand, the guarantees for correctness provided would not be inferior, which distinguishes them from various refactoring tools built into existing IDEs such as Eclipse. I fully expect this new paradigm to revolutionize the way we think about programming through the next decade.