Programmers and end-users often need to solve tricky domain-specific tasks, such as writing text and XML transformations. Despite their complexity, some of these tasks require only a few specialized programming constructs. This aspect opens the opportunity for creating simpler domain-specific tools that do not require the power of general-purpose programming. My research focuses on building programming languages and designing techniques that make the programming of domain-specific tasks simpler, less error-prone, and more efficient.

Leveraging the power of decision procedures

In the last two decades we have seen great improvements in the world of decision procedures, which have led to optimized tools that can efficiently solve many theoretically hard problems. For instance, modern SAT solvers can check the satisfiability of Boolean formulas with hundreds of thousands of variables. Unfortunately, existing decision procedures are still not powerful enough to handle complex tasks such as verifying general-purpose programs. Despite this fact, most of the research in programming languages has focused on these complex tasks, causing these solving technologies to rarely make it to the outside world to meet real-world programmers.

I therefore directed my research to answering the following question: How can we leverage decision procedures to build robust tools that can be used in practice? As a response, I built domain-specific languages that use decision procedures to simplify a wide range of specialized programming tasks, such as programming augmented reality taggers, programming efficient text transformations, designing string encoders, and designing efficient HTML sanitizers. The programming languages I designed satisfy two requirements: 1) they automate a component of the task that is tricky for programmers to handle; and 2) they are simple enough to appeal to their target audience and be used in practice.

These requirements can be seen in action in PrePose, a language for which I filed a patent in collaboration with Microsoft Research. PrePose is a declarative language for describing human postures and gestures for augmented reality (e.g. Lift right arm 30 degrees, or Start in standing position) which uses satisfiability modulo theory (SMT) solvers to automatically answer questions such as: Do the arms ever cross during this sequence of movements? The declarative nature of PrePose abstracts away most of the complexity and makes PrePose easy to use in practice.

Automata-based programming languages

Finite automata enjoy decision procedures that make them a great tool to reason about programs that manipulate strings and trees. Transducers extend automata to model functions that transform strings/trees into strings/trees, and also enjoy many decision procedures. I built three domain-specific languages that use the decision procedures offered by these formalisms to simplify the programming of tedious tasks.

Fast is a domain-specific language for verifying and optimizing programs that manipulate trees over complex domains [DVLM14] (You can try out Fast at http://rise4fun.com/Fast/). We used Fast to 1) check whether programs called HTML sanitizers can ever output potentially malicious HTML trees, 2) prove whether particular augmented reality applications interfere with each other when executed together, and 3) statically optimize functional programs that operate over lists and trees.
DReX is a declarative language for efficiently executing complex text transformations [ADR15]. Programs that transform plain text are ubiquitous, and DReX provides a simple way to write them. The main feature of DReX is its efficiency: every DReX program executes in a linear-time, left-to-right pass over the input string. We showed how complex BibTex transformations could be expressed as simple DReX programs that can process thousands of characters per second.

Bex is a domain-specific language for designing and verifying string encoders and decoders [DV13b, DV13a] (You can try out Bex at http://rise4fun.com/bex/). Cods such as UTF and Base64 are ubiquitous, but their analysis is difficult. In fact, incorrect UTF encodings have been used to bypass security validations. Bex provides specialized constructs to write, analyze, and optimize string encoders and decoders. I used Bex to prove the correctness of UTF and Base64 encoders and generate efficient implementations that are faster than the state-of-the-art ones.

New algorithms and models

My research is driven by practical problems, but I am also interested in answering foundational theoretical questions. This section presents my main theoretical contributions.

Transducer models. During the design of the languages I described above, I developed new models and advance the state-of-the-art decision procedures. In particular, Fast and Bex led me to design two new transducer models and study their properties [DVLM14, DV13a].

I also introduced two novel models for reasoning about XML processing that can benefit from the left-to-right structure of XML trees [AD12, DA14]. For one of the models I devised an algorithm that establishes the first upper bound for checking equivalence for the class of so-called regular tree transformations [AD12].

Algorithms for automata minimization. During my second internship at Microsoft Research, the company was trying to solve the following problem: Given a set of regular constraints, generate uniformly at random a password that satisfies them. My solution made use of symbolic automata, which can describe strings over large alphabets. However, in order for the technique to scale, the automata had to be minimized, and the existing minimization algorithms did not work. Together with Margus Veanes, I devised a new algorithm, which outperforms all existing minimization algorithms [DV14]. In the words of an anonymous POPL reviewer: “This is the first innovative algorithm for the minimization of automata since Hopcroft’s 1971 paper. I will lean far out the window and predict that this will be a seminal paper and trigger a lot of follow-up work.” Microsoft Password Updater uses our algorithm and it is one of the 50 most downloaded tools at Microsoft.

AutomataTutor (used by many Universities in US and Europe)

As a TA for an undergraduate course on automata theory, I was frustrated by how students had to wait days to receive feedback for their homework problems, causing them to forget the rationale behind their solutions and not be able to fully benefit from the feedback. My passion to improve teaching and applying decision procedures led me to design AutomataTutor, a tool for automatically grading automata problems that also provides personalized feedback. In order to identify students’ mistakes, I designed new synthesis algorithms, most notably one to generate English descriptions of the language accepted by the student’s automaton. I used these algorithms to build a grading engine that could also provide actionable feedback. In two separate large user studies we showed that AutomataTutor 1) is more consistent than humans are at grading [ADG+13], and 2) provides feedback that is judged as helpful, increases student perseverance, and can improve problem completion time [DKA+14]. The students’ responses to the tool were also enthusiastic: “I thought the feedback was absolutely excellent”, and “This is a far superior way of learning new things rather than read about something. I hope this way of teaching will be implemented in all schools”. AutomataTutor is now used at Penn, UIUC, UC San Diego, EPFL (Switzerland), and Reykjavik University (Iceland).
Future Work

The main goal of my research is to improve productivity by building tools that simplify tricky, and time-consuming tasks. I envision a world where programming is more efficient, accessible, automatic, and correct, and teaching is more automatic and effective. I believe that decision procedures are essential to achieving this goal, and my research agenda is to identify tasks that can be simplified and made less time-consuming by leveraging such procedures. The following are some of the immediate projects I plan to investigate.

Tools for Personalized Education. Teaching students how to write proofs is a notoriously hard task. Students often ask about where to start or what to do next. As a TA for the graduate course Software Foundations, I learned that the proof assistant Coq helped senior students improve their “proof-writing” skills. However, the tool was confusing to junior students who were new to the concept of proofs, and these students did not benefit from it. I want to design a domain-specific language that contains the features necessary for writing proofs, is simple enough to allow mechanical solutions, and can automatically provide feedback to students. Such a tool would impact the learning experience of thousands of students all over the world.

My interest in automated education ventures outside the classroom. For example, can we use games to teach complex concepts to everyone? Many math and theory problems can be reformulated in a more fun way. For instance, drawing an automaton accepting a given set of strings is the same as drawing a set of intersecting streets that lead people to their destinations after they are given instructions on where to go. Gamifying higher education concepts by leveraging decision procedures intrigues me as it can help educate people at all ages.

Program Synthesis. Lately the problem of synthesizing simple programs from specifications or examples has gained interest, and I want to investigate whether synthesis techniques can be used to simplify domain-specific tasks. For example, writing string coders could be simplified if, given an encoder, we were able to automatically synthesize the corresponding decoder. Another synthesis problem that intrigues me is that of learning automata over structured alphabets. A learning algorithm for this problem would allow us to automatically generate password filters from examples. Solving these theoretically challenging problems can make programming accessible to non-expert users.

Program Monitoring. Programmers often want to monitor runtime properties such as whether the value of a variable remains the same before and after a function is called. To check such a property the user typically writes assertions, which may require modifying the program. However, it turns out that many interesting program properties can be captured and monitored using automata [DA14]. Moreover, since automata enjoy many algebraic properties, they provide a more principled way to describe program monitors. I plan to identify new automata models that can describe more complex program properties, and design declarative languages that can simplify writing and executing program monitors. Advances in this domain can impact many programmers by making the tasks of testing and writing programs more efficient, and scalable.

Crowdsourcing. Recently I collaborated on a new system, CrowdBoost, for crowdsourcing under-specified string filters [CDL+15]. CrowdBoost uses skilled crowd-workers to create initial candidate filters expressed as regular expressions and a different set of unskilled crowd-workers to label positive and negative examples. New improved filters are then produced using genetic programming and automata operations. Can we use similar ideas to crowdsource other computationally hard or under-specified tasks? For example, string sanitizers are short routines that are crucial to preventing cross-site scripting attacks in web applications, yet are difficult to program. If string sanitizers are modeled as transducers, I believe that they can be crowdsourced using similar techniques to those proposed in [CDL+15]. Crowdsourcing can make programming this and other complex tasks less error-prone and more accessible to non-expert users.
Publications


