Research Statement
Mukund Raghothaman

Computer programming is a cognitively demanding activity. I work on building domain-specific languages and tools to simplify this task. My goal is to build theoretically well-understood, rigorously evaluated, and practically useful tools to help programmers in their activities. My research draws from programming languages and theoretical computer science, and covers a broad range of programmer assistance methods: from static analysis and function synthesis tools, to less-clearly defined settings, such as natural language code search.

1 Current Work
1.1 Programming Stream Transformations
As large data streams, such as events from sensors, network logs, webpage clicks, stock tickers, and DNA sequences become more common, there is an emerging need for high-level abstractions to program stream transformations. In my thesis, I introduce the domain-specific language DReX to describe string-to-string transformations, and develop the concept of regular cost expressions (RCE) as a modular way to specify complex aggregation queries over data streams.

String transformations. Our initial objects of interest were simple string transformations, such as reformatting telephone numbers (“123-456-7890” → “(123) 456-7890”), extracting domain names from email addresses (“abc@gmail.com” → “gmail.com”), and string sanitizers and encoding routines in website backends. While regular expressions have been well-accepted by working programmers, and are widely implemented in language libraries, there are fewer facilities for string manipulation. These are typically implemented either by ad-hoc functions, or by Turing-complete tools such as sed and Perl. The first problem is that string manipulation subroutines appear as monolithic blocks of code, with state being explicitly maintained and updated. Second, since the transformation is embedded in a Turing-complete language, most static analysis—a security auditor might want to know, “Does this program ever emit a quotation mark?”—is undecidable.

We observed that a large class of string transformations are instances of simple patterns, such as selecting or deleting substrings with some property, iterating over sub-sequences, or conditional choice. DReX is a modular representation for string transformations, as regular expressions are for sets of strings. There is a small collection of function combinators, such as “if-else”, “split”, and “iter”, corresponding to the natural operations of conditional choice, splitting, and iterating over strings respectively. For example, if the function “get-domain” obtains the domain name from a single email address, then “iter(get-domain)” obtains the domain names from a sequence of email addresses, by separating the string into individual email addresses, extracting the domain name from each address, and concatenating the results.

Function combinators allow us to combine small, easy-to-understand transformations into larger, more complicated transformations. The key advantage is the separation of intent and evaluation: the programmer works with a global view of the string, splitting it into chunks and performing case analysis, while being mostly unconcerned with implementation details such as what state needs to be maintained and how it should be updated. Furthermore, DReX is expressively equivalent to streaming string transducers (SST) [5], a model initially proposed by Rajeev Alur and Pavol Černý to model list-manipulating programs. Transducers are a natural basis for stream transformations because of their well-understood expressive power, appealing closure

\[^1\text{A prototype of DReX is publicly available at www.drexonline.com.}\]
properties, and because of the decidability of analysis questions such as equivalence checking and pre- and post-condition contracts specified as regular languages.

**Quantitative properties.** While DReX is a system for string-to-string, and more generally, stream-to-stream transformations, the idea of using function combinators is also applicable to the quantitative properties of streams. We developed regular cost expressions (RCE) as a natural extension to perform quantitative aggregation over data streams [4]. A data analyst might wish to calculate the average monthly number of website visitors from the access logs. If “#/visits” is the function which takes the server logs for a month and returns the total number of visitors, then the analyst is interested in the function “iter-avg(#/visits)”, the average obtained by splitting the log into individual months, and calculating the total number of views in each.

The main technical challenge in extending DReX to RCEs was in choosing the set of function combinators. While string concatenation is the natural way to combine strings, numerical costs may be combined in many different ways: for example, if $c_1$ and $c_2$ are the costs incurred doing two different activities, then we may be interested both in $\min(c_1, c_2)$ (if we want to perform the less expensive activity) and $c_1 + c_2$ (if we want to do both activities). The question was whether the choice of basic combinators was “sufficient”: the key insight was to generalize RCEs to compute *terms* over data streams rather than individual costs [4, 2]. RCEs are therefore composed of two orthogonal sets of combinators: (a) generalized versions of choice, concatenation, and Kleene-*, from the world of regular expressions, and (b) operators from the cost domain, such as sum, minimum, maximum, average, and median. This decoupling permits evaluation algorithms and expressiveness results to be stated and proved mostly independently of the cost domain.

**Evaluation algorithms.** The most important property of any stream processing system is a fast evaluation algorithm, at most polynomial in the function description $|f|$, and linear in the stream length $|w|$. We showed that if we impose unambiguity requirements on the expressions under consideration, then a remarkably simple single-pass evaluator computes $f(w)$ in time $|w| \cdot \text{poly}(|f|)$ [3]. The algorithm involves simultaneously maintaining multiple potential parse trees of the data stream, and a careful analysis bounds the number of alternatives to at most $O(|f|)$, independent of the length of the data stream seen so far.

### 1.2 Program Synthesis

I am also interested in synthesis, broadly viewed as a programming technique where instead of / in addition to executable code, the programmer describes parts of the program by other techniques, such as by logical specifications, or by input-output examples.

I have collaborated with Abhishek Udupa on the synthesis of distributed protocols [6, 7]. Over the course of developing these synthesis tools and adapting ideas from the research literature, it became clear that there was a core problem which was common to many of these papers: “Synthesize a function $f$ such that for all $x, y, \ldots, \varphi(f, x, y, \ldots)$ holds”, where $\varphi$ was a quantifier-free formula over first-order variables $x, y, \ldots$, and with $f$ appearing as an unknown function. For example, the function returning the maximum of two numbers may be specified as: “Synthesize a function $f$ such that for all $x, y, f(x, y) \geq x \land f(x, y) \geq y \land (f(x, y) = x \lor f(x, y) = y)$.”

Direct comparisons and tool reuse were prevented by instance-specific details, such as differing syntax. From this realization, we developed SyGuS (Syntax-Guided Synthesis), a uniform format for function synthesis, where the unknown function is constrained both by its input-output behavior, and by the syntax of potential solutions [1]. The motivation was two-fold: (a) to act as a uniform format for the consumers and developers of synthesis tools, and (b) to provide a method to empirically benchmark and compare program synthesis tools. Apart from contributing to the formalization of the original SyGuS problem, and writing the SyGuS language description with Abhishek Udupa [8], I also developed stoch, a baseline SyGuS solver based on random walks over expression trees, which went on to place 2nd and 3rd in the first and second SyGuS competitions respectively.

**Code Search.** There is enormous potential in employing open-source code corpuses such as GitHub and BitBucket in programmer assistance tools. On the other hand, the synthesis literature has also focused
on problems with fixed unambiguous specifications. More recently, applications with vague or informal
specifications are receiving renewed attention. I spent two summers at Microsoft Research Cambridge, working
with Youssef Hamadi and Yi Wei on the Bing Code Search tool. Programming languages typically come with
large libraries, and developers often ask API-related questions such as “How do I transmit data over a socket?”
Our goal was to answer these questions with short representative snippets of C# code. Such tools have great
potential to improve programmer productivity, but to be useful, they should be very responsive, answering
queries in a few seconds or less. Our solution [9] was to first use click-through data from the Bing search
engine to map user queries to types and methods of interest. Next, we synthesized code around these APIs
using patterns learned from open-source repositories. We introduced the idea of structured call sequences
to represent these patterns. Structured call sequences are a simple generalization of method invocation
sequences, “new Socket(); Socket.connect(...); Socket.transmit(...); ...”, for example, with if-
branches and while-loops to represent conditional repeated method invocation patterns, and can be easily
extracted from files in the corpus, and can then be readily converted into solution snippets. I believe that
structured call sequences are a fundamental empirical artifact of API design, and can be used in diverse
applications such as detecting code smells. A prototype implementation of our tool was able to answer 30 of
the most commonly occurring C# API-related queries on the Bing search engine, with the first result being
relevant in 70% of the cases, in an average of 1.5 seconds per snippet.

2 Looking Ahead

The overall theme of my research is to improve programmer productivity through a combination of practical
tools and underlying theoretical results. I believe that there is tremendous potential for programmer assistance
tools: constraint solvers such as SAT / SMT solvers can now be reliably used in production code, while the
presence of large amounts of open-source code provides data and benchmarks for empirical tool validation. We
are also at a stage where such techniques would have great impact: there is a large and growing population
who wish to program computers, both expert programmers, and non-expert users wanting to automate simple
tasks. I now outline some broad research outlines I wish to pursue.

Abstractions for stream processing. I plan to investigate the applications of RCEs in monitoring applications,
such as network policies, and streaming data from the increasing number of devices connected to the internet.

The evaluation algorithm for DReX makes a single pass over the stream, and computes its result in time
linear in the stream length. It may, however, consume memory proportional to the length of the input stream.
While this is unavoidable for string-to-string transformations, we can do much better in the quantitative case,
where memory usage is also often tightly controlled. If the set of basic numerical operations is sum, minimum,
maximum, and average, then function evaluation requires $\text{poly}(|w|)$ memory, independent of $|w|$ [4]. There are
exciting opportunities for data-parallel and automatic query approximation. We are currently collaborating
with Sanjeev Khanna and Kostas Mamouras to develop data structures for the approximate evaluation of
RCEs, with the hope that this will enable a significant population of data analysts to leverage results from the
streaming algorithms community.

A major extension to RCEs is to handle multiple input streams. Multiple input streams are traditionally
combined using the database join operation, and is crucial to reconciling events from multiple sources. Multi-
tape transducers provide a compelling alternative to sliding windows, an approach commonly employed by the
literature on streaming databases. I am excited by several questions in this area: What is the class of queries
expressible by multi-tape transducers? What are the most convenient operators and abstractions to access
these features? What static analysis questions are decidable in these models?

Monitoring cyber-physical systems for robustness. Another intriguing application is in online monitoring
of traces with respect to signal temporal logic (STL) properties, and was recently suggested by Jyotirmoy
Deshmukh. Signal temporal logic is an extension of classical linear temporal logic to hybrid systems, and
is widely used in the industry, such as by engineers working on automotive control. As opposed to the “all-or-nothing” nature of LTL, engineers are often interested not just in boolean values, but also in robustness estimates: an engineer who asserts “The engine speed will never rise above 5000 rpm” wishes not just to confirm this claim, but also in the maximum engine speed ever attained. STL naturally admits a robustness metric for signal traces, and this value can be computed by a postmortem algorithm in linear time. Can the robustness value be computed by an online algorithm?

In a broader sense, there are deep connections between the memory constraints imposed on big-data streaming algorithms, and the amount of memory available to cyber-physical systems. Can algorithmic insights and programming abstractions from one field be profitably exploited in the other?

Static analysis of string manipulating programs. While DReX supports decidable static analysis, there is a large body of string manipulating programs written in Turing-complete and difficult-to-analyse languages such as sed and Perl. These range from Bash scripts which mangle file names to security-critical string sanitizer scripts. While there is a need for static analysis tools for all these programs, most current literature focuses on individual languages, because language-specific details overwhelm the potential for portable analysis. I am collaborating with Arjun Radhakrishna and Loris D’Antoni to build black-box tools for the verification of string manipulating programs. The idea is to exploit the fact that DReX expressions are easily enumerable, and automatically synthesize and validate expressions consistent with a set of input-output examples.

Computational techniques for program synthesis. There are many outstanding problems in function synthesis, both in developing better (SyGuS) back-ends, and in exploring new front-end applications. From the results of both the SyGuS competitions, it is clear that a lot of progress is needed before SyGuS solvers become as robust as SAT / SMT solvers are today. Furthermore, most of the competition benchmarks test single-function constraints. From my experience with protocol synthesizers, multi-function constraints such as “Synthesize functions f, g, such that for all x, y, f(x, y) + g(x, y) = x + y” are very common, but it is precisely these “jointly-constrained” problems that current solvers have the hardest time solving. Finally, most current solvers are instantiations of the counter-example guided inductive synthesis (CEGIS) loop, but the theoretical properties of CEGIS are not well-understood. I plan to investigate CEGIS-free techniques for the function synthesis problem, with an eye towards improving performance on multi-function benchmarks.

We can usually assume, and most often verify the output of function synthesizers, but only against the input constraints provided. Many constraints, particularly those relating to performance, are hard to describe, and omitted from the specification submitted to the synthesizer. Even more worrisome is the possibility of unintentionally omitted constraints. To what extent can the user trust the output of the function synthesis tool? Manual inspections of the output might be one solution, but what if the synthesized solutions are too large or complicated to be manually audited?

There is a large amount of unmaintained and hard-to-maintain software written in languages such as COBOL. There are also simple data handling subroutines embedded in Turing-complete languages which could be more conveniently written in domain-specific languages, or ported to use more modern libraries. This problem—of synthesizing specifications and equivalent representations in a different framework—is a problem for which techniques such as SyGuS and code repository mining are uniquely suited. Can we synthesize declarative specifications from legacy code?

References


