Tab 1

Types4Strings Project Ideas

If you are interested in any of these projects or have ideas of your own, please do not hesitate to contact us! You can send an email to either michael.schroeder@tuwien.ac.at or juergen.cito@tuwien.ac.at. Please attach a recent course transcript (Sammelzeugnis) so that we can get an idea of your background.

Regex Visualization Tool

Background	
Motivation	
Outcomes	Web-based tool that interfaces with our \`regex-algebra\` Haskell library or the \`regex\` CLI tool. It should support at least the following operations of the library/tool: - Checking equality and inclusion of regular expressions, showing counter-examples in the negative case - Intersection and complement of regular expressions - Simplification of regular expressions - Converting regular expressions between common representations, e.g., POSIX and PCRE
Approach	1. Extend the regex-algebra package or the regex CLI tool to support the envisioned interactions, if necessary.

	2. Implement the web interface to visualize and manipulate regular expressions in an algebraic fashion. 3. Make this tool publicly available. This requires designing a deployment infrastructure that is both secure and limits resource usage.
Scope	**Project in CS**

Parser Dataset Annotation Tool

Background	Ad hoc parsers are pieces of code that use common string functions like split, trim, or slice to effectively perform parsing. Ad hoc parsing is ubiquitous—yet poorly understood. As part of the TYPES4STRINGS project, a number of research groups are investigating various approaches to analyze these kinds of parsing programs in various ways. To be able to compare these different approaches, we are currently compiling a large-scale benchmark dataset of real-world ad hoc parsers. To be useful as a benchmark, such a dataset must include not only the ad hoc parser code itself, and various amounts of metadata, but also, crucially, the *ground truth* associated with each ad hoc parser: a regular expression representing the input language that the parser accepts (if the parser is regular and such an expression exists).
Motivation	Manually annotating our vast dataset with ground truth is tedious and error-prone but ultimately unavoidable. However, we can improve the annotator’s workflow, and provide some amount of quality assurance, by providing them with a convenient tool that automates as much as possible.
Outcomes	TBD
Approach	- possible starting point: https://github.com/ductnguyen12/code-annotation/wiki/Snippet
Scope	**Project in CS**

More powerful integer abstract domain for Panini

Status	**Assigned**. *If you are interested in a similar project, please write to us anyway!*
Background	Abstract interpretation is a way to soundly approximate the semantics of computer programs. Key to this technique are efficient implementations of *abstract domains*, i.e., representations of the potentially infinite sets of values that variables might assume during possible executions of a program. For example, the value of x in the expression x > 5 can be abstractly represented as the infinite interval [6,∞].
Motivation	The Panini grammar inference system makes heavy use of abstract interpretation to infer regular expressions from predicate logic constraints representing parser programs. It currently uses an interval sequence domain to represent infinite sets of integers, allowing us to abstract constraints such as $x > 0 \wedge x \neq 5$ into simpler forms like $x = [1, 4 \| 6, \infty]$. However, more complex constraints, e.g., congruence classes such as x mod 2 = 0, cannot currently be efficiently abstracted. In the context of grammar inference, this can lead to under-approximations for certain parsers. Thus, we would like to extend Panini's integer abstract domain to cover a larger subset of $\wp(\mathbb{Z})$, with the specific goal to support inference of regular language grammars.
Outcomes	A new integer abstract domain for Panini that is strictly more powerful and enables inference of a larger set of regular languages.
Approach	1. Survey the literature on integer abstract domains and related topics 2. Determine what (combinations of) domains would be useful for regular grammar inference 3. Implement these domains in Haskell as part of Panini 4. Evaluate the domains with detailed case studies
Scope	**Master Thesis** or **Project in CS** (implementation only)

Panini frontend for JavaScript / TypeScript
Panini frontend for C
Panini frontend for Java

Background	The Panini grammar inference system is built around a common intermediate representation of parser programs. Parsers written in high-level source languages (e.g., Python) are first translated to this common IR, which is then used to infer the parser's grammar.
Motivation	Adding a new language frontend significantly expands the reach of Panini and brings its capabilities to a whole new set of users and domains.
Outcomes	● A new language frontend for Panini. ● A benchmark dataset for Panini in the new language.
Approach	1. Familiarize yourself with Panini and the built-in Python frontend. 2. Formally define a translation from the source language to Panini. This includes Panini axiomatizations of the source language's standard library functions. 3. Implement this translation either in the source language or in Haskell as part of Panini. 4. Evaluate the new frontend on an extensive set of test parsers in the source language. These should include a. the OOPSLA25 Panini benchmark dataset, translated into the source language b. New test programs that exercise particular features of the source language
Scope	TBD

Re-implement Panini Python frontend in Python

Background	The Panini grammar inference system is built around a common intermediate representation of parser programs. Parsers written in high-level source languages (e.g., Python) are first translated to this common IR, which is then

	used to infer the parser's grammar. Currently, Panini's Python frontend is implemented in Haskell as part of Panini.
Motivation	Implementing the Python frontend in Python could allow us to more easily and closely track the Python AST and more quickly adapt to changes in new language versions. It might also make it easier to interface with larger parts of the Python ecosystem.
Outcomes	TBD
Approach	TBD
Scope	TBD

Modular Panini SMT solver backend

Background	The Panini grammar inference system uses the Z3 SMT solver during refinement type inference.
Motivation	We want to improve SMT solving performance, both in terms of runtime and capabilities.
Outcomes	● Modular SMT solver interface in Panini ● Seamless switching between solver backends (at least Z3 and CVC5) ● Reduced solver overhead through inter-process communication
Approach	1. Implement new SMT solver subsystem in Haskell as part of Panini 2. Evaluate performance differences and solver capabilities
Scope	**Project in CS** or **Bachelor Thesis** (with extended evaluation)

Assorted ideas / Miscellania

- Take ownership of (or fork) the language-python Haskell package and add support for the latest Python version

- Extend Panini Python frontend to support more syntax (might need Jakob’s extensions first)
- Merge Jakob’s extensions to support datatypes and polymorphism
- Extend the regex-algebra library with optional provenance annotations that are preserved under the various operations and simplifications (provenance calculus?)
- Add provenance information throughout the whole Panini pipeline
- Panini web interface that makes use of provenance info
- Panini VS Code plug-in
- Add TreeSitter grammar for the Panini language