Teaching Type Systems Implementation with Stella, an Extensible Statically Typed Programming Language

Nikolai Kudasov joint with Abdelrahman Abounegm and Alexey Stepanov

Trends in Functional Programming in Education (TFPIE), January 9th, 2024

Lab of Programming Languages and Compilers

INNOPOLIS UNIVERSITY

Context

Students and course format:

1. 3rd year undergraduate students under "Software Development" track
2. 60–90 students (2–3 groups)
3. an intensive "block" course (second half of semester, 2 lectures + 2 labs per week)
4. follows an introductory "Compilers Construction" course based partially on classic textbooks (Appel 2004; Muchnick 1998; Wirth 1996)

Intended learning outcomes:

1. Understand what a type system is and what properties one might expect from it
2. Understand, be able to follow and implement typechecking procedure(s)
3. Reason about program behaviour with types
4. Understand challenges of mixing some of the type system features
5. (extra) Understand some ideas for compiling lazy functional programs

Syntax of λ-calculus vs Python/C/Java

First iteration of the course followed closely TaPL (Pierce 2002), however, students were struggling to absorb coding exercises based on typed λ-calculi.

We noticed that students struggle with the *syntax* of λ-calculus, but managed to internalize and explain the *semantics* better when examples are translated into equivalent programs in Python, C++, or Java.

So, in the second iteration of the course, we have decided to **replace the syntax** with something that students can potentially absorb better.

The result is the Stella language[1].

[1] https://fizruk.github.io/stella/

Stella Core

Stella Core is a minimalistic expression-based purely functional programming language:

1. pure single-parameter top-level named functions
2. built-in `Nat` and `Bool` types with corresponding functions and literals
3. first-class functions
4. Rust-inspired syntax

```
// sample program in Stella Core
language core;

fn increment_twice(n : Nat) -> Nat {
  return succ(succ(n))
}

fn main(n : Nat) -> Nat {
  return increment_twice(succ(n))
}
```

Stella Language Extensions

Stella features a number of language extensions, most of which follow certain sections from TaPL (Pierce 2002). The idea is that most extensions are small enough to be completed as a part of a coding assignment.

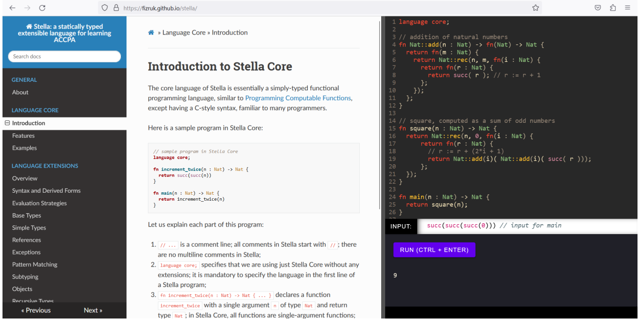
```
language core;

extend with #records, #structural-subtyping;

fn getX(r : {x : Nat}) -> Nat {
  return r.x
}

fn main(n : Nat) -> Nat {
  return getX({x = n, y = n});
}
```

An Incomplete List of Stella Extensions

Feature	Full	Partial	Full, %	Partial, %
Records	46	0	94%	0%
Pairs	45	0	92%	0%
Unit type	44	0	90%	0%
Sequencing	43	0	88%	0%
References	42	0	86%	0%
Sum types	39	0	80%	0%
Errors	39	0	80%	0%
Subtyping for records	39	0	80%	0%
Tuples	38	3	78%	6%
Universal types	17	21	35%	43%
Top and Bot types	14	8	29%	16%
Exceptions with a fixed type	11	0	22%	0%
Exceptions with an open variant type	10	0	20%	0%
`letrec`-binding	9	0	18%	0%
Subtyping for variants	9	0	18%	0%
Variants	7	0	14%	0%
`let`-binding	7	2	14%	4%
Structural patterns	3	21	6%	43%

Documentation and Playground

The website[2] features documentation and a live Playground so that students can check different programs and compare the canonical implementation against theirs.



[2] https://fizruk.github.io/stella/

Implementation Templates

We encourage students to use different implementation languages. To support that, we provide implementation templates[3] in C++, Java, Kotlin, OCaml, TypeScript, Swift, Rust, Go, and Python.

The templates feature a parser, a set of types for AST, an AST traversal skeleton, and a pretty-printer. A big part is generated via BNF Converter (Forsberg and Ranta 2004) and/or ANTLR. Importantly, students do not have to work with the parser, they work with the AST for the *full language* and are allowed to simply ignore unsupported parts.

From our experience, once the project template loads in an IDE(s) that students are used to (usually, VS Code, IntelliJ IDEA and other JetBrains' IDEs), *setup takes virtually no time* for them.

[3] see https://github.com/IU-ACCPA-2023

Course Structure

The most recent iteration of the course featured several blocks:

1. **Simple Types**: Stella Core, `Unit`, pairs, sum types
2. **Normalization and Recursive Types** (theoretical interlude)
3. **Imperative Objects**: sequencing, mutable references, structural subtyping;
4. **Type Reconstruction and Universal Types**: constraint-based type inference, System F and Hindley-Milner type systems;
5. **Runtime for Lazy Functional Languages**: STG language (Jones 1992)

Due to the time limitations for the course (half-semester), the amount of material is restricted, so we do not explore topics such as substructural type systems, dependent types, and higher-order types.

Results

Overall, using Stella language in place of typed λ-calculi was appreciated by the students and we have seen significant improvement in students' performance[4].

We had a one week delay at the beginning of the course, since we initially provided Makefile-based project templates, but most students experienced issues with it (also due to lack of experience with such setups). We do not expect similar delays in the future.

Students struggled most with implementation of *Universal Types* (due to name captures!) and *Structural Patterns* (typically, due to mutable state and Visitor pattern implementations).

[4] in terms of completed coding assignments

Conclusion

We have implemented a half-semester course, focused around the study and implementation of type systems, supported by a special language Stella. Overall, we think of our experience as positive, although many improvements are possible:

1. Support more of standard extensions: nominal types, bounded universal quantification, `throws`-annotations, and more;
2. Add more extensions related to operational semantics (to explore compiler backends for functional languages);
3. Improve quality and automation of tests.
4. Make it possible to implement Stella in Stella?

Thank you!

- Appel, Andrew W. (2004). ***Modern Compiler Implementation in ML*.** USA: Cambridge University Press. isbn: 0521607647.
- Forsberg, Markus and Aarne Ranta (2004). **"BNF Converter".** In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: Association for Computing Machinery, pp. 94–95. isbn: 1581138504. doi: 10.1145/1017472.1017475. url: https://doi.org/10.1145/1017472.1017475.
- Jones, Simon L. Peyton (1992). **"Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine".** In: *J. Funct. Program.* 2.2, pp. 127–202. doi: 10.1017/S0956796800000319. url: https://doi.org/10.1017/S0956796800000319.

Muchnick, Steven S. (1998). ***Advanced Compiler Design and Implementation***. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. isbn: 1558603204.

Pierce, Benjamin C. (2002). ***Types and programming languages***. MIT Press. isbn: 978-0-262-16209-8.

Wirth, Niklaus (1996). ***Compiler Construction***. USA: Addison Wesley Longman Publishing Co., Inc. isbn: 0201403536.