Automatically Constructing a Type System from the Small-Step Semantics

Ștefan Ciobâcă and Vlad Andrei Tudose*

Faculty of Computer Science,
"Alexandru Ioan Cuza" University,
Iași, Romania
stefan.ciobaca@info.uaic.ro

Abstract

We describe preliminary work suggesting that most typing rules for a programming language can be obtained automatically from its operational semantics. Instead of going the usual way, to first define the semantics and the type system, and then show progress and preservation, we start from the semantics and construct a type system that satisfies progress and preservation by construction. We have tested our approach on simple lambda calculi and we have constructed a Haskell prototype that implements our algorithm.

1 Inferring Typing Rules

Usually, type soundness proofs for programming languages are quite straightforward. This is not too surprising, since most typing rules are quite similar to the operational semantics rules, when looking at them from a meta-syntactic point of view. We conjecture that most typing rules for a language can be generated automatically from the operational semantics. Type soundness should then follow immediately, by construction.

Imagine a programming language with the following syntax:

```
<t> ::= true | false | if <t> then <t> else <t> | 0 |
succ <t> | pred <t> | isZero <t>
<nv> ::= 0 | succ <nv>
<bv> ::= true | false
<v> ::= <nv> | <bv>
```

and the folllowing associated operational semantics:

$$\texttt{if true then } \langle t_2 \rangle \texttt{ else } \langle t_3 \rangle \rightarrow \langle t_2 \rangle \qquad \texttt{if false then } \langle t_2 \rangle \texttt{ else } \langle t_3 \rangle \rightarrow \langle t_3 \rangle$$

$$\frac{\langle t_1 \rangle \rightarrow \langle t_1' \rangle}{\texttt{if } \langle t_1 \rangle \texttt{ then } \langle t_2 \rangle \texttt{ else } \langle t_3 \rangle \rightarrow \texttt{if } \langle t_1' \rangle \texttt{ then } \langle t_2 \rangle \texttt{ else } \langle t_3 \rangle} \qquad \frac{\langle t_1 \rangle \rightarrow \langle t_1' \rangle}{\texttt{succ } \langle t_1 \rangle \rightarrow \texttt{succ } \langle t_1' \rangle}$$

$$\texttt{pred 0} \rightarrow \texttt{0} \qquad \texttt{pred (succ } \langle t_1 \rangle\texttt{)} \rightarrow \langle t_1 \rangle \qquad \frac{\langle t_1 \rangle \rightarrow \langle t_1' \rangle}{\texttt{pred } \langle t_1 \rangle \rightarrow \texttt{pred } \langle t_1' \rangle} \qquad \texttt{isZero 0} \rightarrow \texttt{true}$$

$$\texttt{isZero (succ } \langle t_1 \rangle\texttt{)} \rightarrow \texttt{false} \qquad \frac{\langle t_1 \rangle \rightarrow \langle t_1' \rangle}{\texttt{isZero } \langle t_1 \rangle \rightarrow \texttt{isZero } \langle t_1' \rangle}$$

*The work reported in this paper was performed while the second author was a Master's student at the Faculty of Computer Science.

What could be the typing rules for the language? Next, we will assume that all typing rules have the following shape, for all operators `op` of the language:

$$\frac{\langle t_1\rangle : \langle T_1\rangle \qquad \langle t_2\rangle : \langle T_2\rangle \qquad \ldots \qquad \langle t_n\rangle : \langle T_n\rangle}{\texttt{op}\ \langle t_1\rangle\ \langle t_2\rangle\ \ldots\ \langle t_n\rangle : \langle T\rangle} \quad (Name)$$

Assume that we know the typing rules for `true` and `0` are:

$$\frac{}{\texttt{true} : \texttt{Bool}} \qquad \frac{}{\texttt{0} : \texttt{Nat}}$$

and suppose that our typing system has the preservation property. When we look at the rule for `isZero`:

`isZero 0` $\rightarrow$ `true`

it is immediate that isZero 0 must also be of type `Bool` (otherwise preservation would not hold). Furthermore, by our assumption on the set of operators `op` of the language and since `0 : Nat`, the isZero operator must satisfy the following typing rule:

$$\frac{\langle t_1\rangle : \texttt{Nat}}{\texttt{isZero}\ \langle t_1\rangle : \texttt{Bool}}$$

By reasoning similar to the above, we can infer the expected typing rules for all other language operators, as presented in [**?**]. Any well-typed term in the resulting type system does not get stuck. We have implemented and shown that our algorithm for transforming the operational semantics into typing rules is sound, in the sense that the resulting typing system has progress and preservation. Furthermore, we have extended our approach to handling more complex typing rules, which have the following shape:

$$\frac{\Gamma_1 \vdash \langle t_1\rangle : \langle T_1\rangle \qquad \Gamma_2 \vdash \langle t_2\rangle : \langle T_2\rangle \qquad \ldots \qquad \Gamma_n \vdash \langle t_n\rangle : \langle T_n\rangle}{\Gamma \vdash \texttt{op}\ \langle t_1\rangle\ \langle t_2\rangle\ \ldots\ \langle t_n\rangle : \langle T\rangle} \quad (Name)$$

This allows us to infer the typing rules for simple lambda calculi presented in Curry style.

Our prototype implementation, written in Haskell, can be accessed at `http://profs.info.uaic.ro/~stefan.ciobaca/types2017`. The program takes as input the files `syntax.txt` (describing the syntax of the language in a BNF-like format), `eval_rules.txt` (describing the small-step SOS of the language), `type_syntax.txt` (describing the syntax of the types of the language) and `typing.txt` (describing the basic typing rules, such as `0 : Nat`) and outputs the rest of the typing rules.

2 Discussion

This work could simplify type system design for programming languages, as there would be no need to prove progress and preservation. Our approach cannot currently handle sub-typing, polymorphism or exception handling and we leave these features for further work. The final goal is to start from a formal semantics, for example reified from a Coq development, assume progress and preservation, and automatically derive a mechanically proven sound-by-construction typing system.

References

[1] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.