A Language for the Specification and Efficient Implementation of Type Systems

Pascal Wittmann
TU Darmstadt, Germany

Abstract

Type systems are important tools to ensure partial correctness of programs, to establish abstractions and to guide the programmer in the development process. However, there is currently a lack of established tools supporting the development of type systems. Tools like lexer and parser generators. We introduce a declarative specification language for type systems, that allows to specify type systems in a natural deductive style. We generate two products from a specification: A first-order formula representation to facilitate the use of automated theorem provers and an efficient type checker. Both results aim to make the development cycle for type systems faster and to narrow the gap between theory and practice.

1. Motivation

Type systems ensure partial correctness of programs. In other words, they try to ensure that programs have meaning in the sense of the semantics of the programming language. The type systems we focus on are static type systems and can also be thought of as a static approximation of the program semantics. Besides ensuring partial correctness, type systems are means to establish abstractions, to enforce adherence to these abstractions and they can serve as documentation. All in all type systems can help to develop software more efficiently (cf. [5] and [3]).

Type systems are useful tools if they fit to the programming language and the application scenario. To ensure this it makes sense to adapt and modify existing type systems or create new specialized type systems. Those specializations can lead to better error messages, more expressive type systems and the detection of more errors [9]. Currently there are, to our best knowledge, no established tools that generate type checkers from a declarative specification. Such generators could make the development of type checkers faster and less error prone. Those generators would fit well into the language development workbench besides the long established lexer and parser generators.

The other advantage of generating a type checker from a declarative specification language is that the specification is close to the formal description of type systems in text books. Thus it is possible to adapt results from text books to a specification and those will apply to the generated type checker as well. Such a comparison is not possible with traditional type system implementations. Declarative specifications also allow to specify type systems from other areas, like language-based security, close to their formalizations.

To have even more confidence into properties of type systems the translation of a specification into first-order formulas can be used to conduct proofs of these properties. As the proofs of propositions about type systems change when the type system changes, it is desirable to make most of these changes automatically. Automated theorem provers can try to conduct those proves automatically and if the proof search fails one can give hints for proof search. This establishes a direct connection between the specified type system, the proofs and the generated type checker.

2. Research Problem

The research problem we are tackling is to create a tool that allows to create an efficient type checker from a declarative specification and that narrows the gap between formal reasoning about type systems and their implementations. This problem splits up in smaller problems.

The first problem is the design of a declarative specification language that is close to the text-book formalisms of type systems. This language should make it easy to use existing syntax definitions of realistic programming languages and the difference between typing rules on paper and in the specification language should be small.

The next problem is to create a good first-order formula representation of specifications. This representation should be designed such that it suits automated theorem proving and thus supports a fast prove search. It should also be investigated to which extend theorem provers for first order logic can be used for type checking.

The main problem is the generation of an efficient type checker. The generated type checker should be able to cope with non-syntax directed typing rules in an efficient manner. In this context it should be investigated whether facts proven by an automated theorem prover can be exploited for the type checker generation.

3. Related Work

Early approaches used to generate type checkers were not particularly designed for type checker generation. For example the Synthesizer Generator [6] which uses attributes grammars and the ASF+SDF Meta-Environment [10] base on conditional term rewriting. Later approaches use similar techniques, for example TCG [1] which uses inference rules, but are specialized for the generation of type system. Those approaches have a usually lower performance than handcrafted implementations, because they implement the control-flow implied by the used formalism.

A recent approach (TyC [4]) has focused on the generation of type checkers for object-oriented programming languages. It

```
module example
imports common
language simply-typed-lambda-calculus
meta-variables  Term "~" { Type Exp }
                Ctx "$" { Context }
                Id "%" { ID }
contexts Context := ID{I} x Type{O}
judgments Context{I} "|-" Exp{I} ":" Type{O}.
rules
%x : ~T in $C
============== T-Var
$C |- %x : ~T

(%x : ~T ; $C) |- ~t : ~T
================================ T-Abs
$C |- \ %x : ~T . ~t : ~T -> ~T

$C |- ~t1 : ~T11 -> ~T12
$C |- ~t2 : ~T2
======================== T-App
$C |- ~t1 ~t2 : ~S
```

Figure 1. Specification of the type system for the simply typed lambda calculus.

provides a framework to build efficient type checkers for object-oriented languages (without polymorphism) and uses for this purpose, in most parts, normal program code that implements an interface to specify the type system. Thus type systems cannot be specified declaratively with this approach.

Ott [7] is an other approach that generates from a specification language code for proof assistants and LaTeX, it tries to reduce the gap between the hand-written on-paper proofs and machine checkable proof. Ott focuses on reducing the boilerplate when formalizing type systems, it does not attempt to generate a type checker.

4. Approach

Our approach combines ideas from the related work presented above. We design a high level declarative specification language that is close to text-book formalizations of type systems and generate first-order formulas and efficient type checkers from it.

We use the language workbench Spoofax [2] for the implementation of the specification language and the generators.

Specifications are organized in modules and contain declarations of meta-variables, contexts, judgments, rules and test-cases. Figure 1 gives and impression of a specification for the simply typed lambda calculus.

We transform a specification into first-order formulas in the TPTP [8] format. The use of the TPTP format allows to use a variaty of different automated theorem provers. Typing rules are transformed roughly into the following format, where $p_1 \dots p_n$ are the premisses, c is the conclusion and $free$ collects free variables:

$$\forall v \in free(p_1, \dots, p_n, c).p_1 \wedge \dots \wedge p_n \implies c \qquad (1)$$

In this transformation process measurements are taken to ensure that variables are correctly quantified and valid TPTP formulas are generated.

The generated type checker consists of a constraint generator and a constraint solver. We have chosen constraint solving as a basis for the type checker, because it allows the generation of fast type checkers and is not bound to certain class of type systems. To obtain a fast constraint generator the structure of the rules should be analyzed, also with the help of the formula generation and automated reasoning. A possible approach for dealing with non-syntax-directed rules is for instance to check whether two rules commute, to delay to application of general rules as long as possible.

5. Contributions

This work has three main contributions: A declarative specification language that allows to specify arbitrary type systems close to text-book formalizations, a tool that transforms specifications into first-order formulas in the TPTP format, and finally a tool that generates an efficient type checker that can cope with non-syntax-directed rules and uses automated reasoning to optimize the resulting type checker. This reduces the amount of work to implement type systems and reduces the gap between theory and practice.

We have working prototypes of the specification language and the formula generation and tested those with type systems of languages like PCF and SystemF and with a type system from information flow security. The generation of the type checker is work in progress.

References

[1] Holger Gast. *A generator for type checkers.* PhD thesis, Eberhard Karls University of Tübingen, 2005. http://d-nb.info/977024180.

[2] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.

[3] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM.

[4] Francisco Ortin, Daniel Zapico, Jose Quiroga, and Miguel Garcia. Automatic generation of object-oriented type checkers. *Lecture Notes on Software Engineering*, 2(4), 2014.

[5] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 212–222, New York, NY, USA, 2014. ACM.

[6] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 42–48, New York, NY, USA, 1984. ACM.

[7] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[8] Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. Tstp data-exchange formats for automated theorem proving tools, 2004.

[9] Peter Thiemann. Programmable type systems for domain specific languages, 2002.

[10] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The asf+sdf meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 365–370, London, UK, UK, 2001. Springer-Verlag.