Program Logics for Mechanizing Typechecking

DENIS CARNIER, Vrije Universiteit Brussel (Student author, ACM: 7577148, Category: MSc. student)
STEVEN KEUCHEL, Vrije Universiteit Brussel (Research advisor)

1 INTRODUCTION

Many modern programming languages are statically typed. Every piece of data is classified by its type, and at compile time, programs are checked for consistent usage through a process known as typechecking (TC). Like most complicated systems, good software engineering practices (e.g. separation of concerns via phase separation, computational abstraction through monads and applicatives, etc.) can help language implementors write maintainable TC code.

Designing sound and decidable type systems is hard. Therefore, researchers formally study type systems and TC algorithms and strive to mechanize them in proof assistants. Unfortunately, mechanized implementations usually do not follow the same software engineering principles. Exceptions to that rule exist, e.g. Silva et al. [11] implement monadic TC in Coq, but we are unaware of any mechanization that combines a monadic implementation with phase separation. Moreover, practical concerns like elaboration or type reconstruction are usually not covered.

Our goal is to develop a general approach to mechanizing TC, that uses abstraction via monads, employs phase separation and can also produce multiple outputs, e.g. for type reconstruction.

2 BACKGROUND AND STATE OF THE ART

We revisit modern methods to implement TC in Sect. 2.1 and briefly discuss background on the verification of monadic code in Sect. 2.2.

2.1 Implementation

Traditional TC algorithms, like algorithms $\mathcal{J}$ and $\mathcal{W}$ [7] or Morris' algorithm [8], solve equality constraints as they come up. An important realization, first described by Wand [16], is that TC can be split into two distinct phases. During the first phase (a), the algorithm generates constraints for the entire input program without solving them. Afterwards, in the second phase (b), these constraints are submitted to a solver to produce an output. This phase-separated approach facilitates the re-use of a constraint language and its solver for different object languages and type systems.

Central to type inference for many languages is the notion of unification variables (uvars): placeholders for yet unknown types, which a unifier will replace with concrete types during constraint solving. These become necessary when not all type inputs are available for a recursive call, e.g. the type of a let-bound program variable. During the generation of constraints, these uvars correspond to an existential quantification of a type. Another ubiquitous constraint is equality, e.g. in a function application the type of the domain of the function must be equal (during solving this means unifiable) with the type of the passed argument. More complex type systems may require more forms of constraints, e.g. subtyping, which we do not cover here.

Practical implementations also perform *elaboration*, a distinct third phase (c) that performs a compilation or desugaring step during TC that is usually type-directed. Examples include transforming HM terms into System F terms [9], dictionary translation of type classes [15], and elaborating modules to records [10]. For all intents and purposes, elaboration is an additional output that is generated during TC. In the remainder of this paper, we only cover the simple case where that additional output is a type reconstructed term in the same object language.

$$
\begin{aligned}
&M &&: \mathit{Type} \to \mathit{Type}\\
&\mathit{fail} &&: \forall a.M\ a\\
&\mathit{assert} &&: \mathit{Ty} \to \mathit{Ty} \to M\ ()\\
&\mathit{exists} &&: M\ \mathit{Ty}
\end{aligned}
$$

Fig. 1. Typecheck monad interface

$$
\begin{aligned}
&\sigma, \tau \in \mathit{Ty} &&::= \mathit{int} \mid \mathit{bool}\\
&e \quad \in \mathit{Exp} &&::= x \mid n \mid b \mid \textbf{if } e \textbf{ then } e \textbf{ else } e\\
&&&\quad \mid e + e \mid e < e \mid \textbf{let } x = e \textbf{ in } e\\
&x \quad \in \mathit{Var} \quad n \in \mathit{Int} \quad b \in \mathit{Bool}
\end{aligned}
$$

Fig. 3. Grammar of AE+Let

$$
\begin{aligned}
&\mathit{WP}, \mathit{WLP} : M\ A \to (A \to \mathit{Prop}) \to \mathit{Prop}\\
&\mathit{WP} \quad (\mathit{return}\ x) \quad P \leftrightarrow P\ x\\
&\mathit{WP} \quad (m \gg\!= f) \quad P \leftrightarrow \mathit{WP}\ m\ (\lambda x.\mathit{WP}\ (f\ x)\ P)\\
&\mathit{WP} \quad \mathit{fail} \quad P \leftrightarrow \mathit{False}\\
&\mathit{WP} \quad (\mathit{assert}\ \sigma\ \tau) \quad P \leftrightarrow \sigma = \tau \wedge P\ ()\\
&\mathit{WP} \quad \mathit{exists} \quad P \leftrightarrow \exists \tau.P\ \tau\\
&\mathit{WLP}\ \mathit{fail} \quad P \leftrightarrow \mathit{True}\\
&\mathit{WLP}\ (\mathit{assert}\ \sigma\ \tau) \quad P \leftrightarrow \sigma = \tau \to P\ ()\\
&\mathit{WLP}\ \mathit{exists} \quad P \leftrightarrow \forall \tau.P\ \tau\\
&(\forall x.P\ x \to Q\ x) \to \mathit{WP}\ m\ P \to \mathit{WP}\ m\ Q
\end{aligned}
$$

Fig. 2. Typecheck monad program logic

2.2 Verification

Verification means that we want to reason about the correctness of an implementation against a specification. We are primarily interested in showing the equivalence between a TC algorithm and a declarative type system. The equivalence can be split into two directions: soundness and completeness. We first discuss our approach to verification and come back to these properties after.

Program logics define a set of reasoning rules to establish the correctness of (imperative) programs. For instance, Hoare logic [2] uses judgements of the form $\{\!\{\ P\ \}\!\}\ s\ \{\!\{\ Q\ \}\!\}$ that specify a precondition P and postcondition Q for a statement s. Using a weakest-precondition transformer WP [1], this triple can be stated equivalently as $P \to \mathrm{WP}\ s\ Q$. Traditionally, program logics work with the syntax of a language, but they can also be defined for monads [5, 12–14]. This is already exploited in [11], which use a *Hoare-State-Exception* monad to implement and verify $\mathcal{W}$. We use weakest preconditions for monads instead, which have been extensively studied in the context of Dijkstra monads [5, 12, 13].

Using program logics, we can state the completeness of a (hypothetical) monadic typechecker check as the equivalent *total correctness* judgements

$$\{\!\{\ \Gamma \vdash e : \tau\ \}\!\}\ \mathrm{check}\ \Gamma\ e\ \tau\ \{\!\{\ \mathrm{True}\ \}\!\} \qquad \Gamma \vdash e : \tau \to \mathit{WP}\ (\mathrm{check}\ \Gamma\ e\ \tau)\ (\mathrm{True}) \tag{1}$$

i.e. under the precondition $\Gamma \vdash e : \tau$ the TC succeeds. Soundness is expressed by the judgments

$$[\![\ \mathrm{True}\ ]\!]\ \mathrm{check}\ \Gamma\ e\ \tau\ [\![\ \Gamma \vdash e : \tau\ ]\!] \qquad \mathrm{True} \to \mathit{WLP}\ (\mathrm{check}\ \Gamma\ e\ \tau)\ (\Gamma \vdash e : \tau) \tag{2}$$

for which we use *partial correctness* Hoare triples or *weakest liberal preconditions*. They express that under the precondition True, if the TC succeeds then $\Gamma \vdash e : \tau$ holds.

3 APPROACH

In this section, we present our approach of implementing and mechanizing monadic, phase-separated TC with elaboration. We discuss TC without and with unification separately.

3.1 Without unification

For implementing TCs, we use an abstract interface of a typechecking monad that, besides return and bind, supports multiple operations which can be found in Fig. 1. First, the *fail* allows us to model explicit failure, for example when a referenced program variable is unbound. Next, the *assert* operation ensures that two types are equal. We discuss the *exists* operation in the next section.

With this interface, we can implement TC for languages that do not rely on unification. For instance, for the language of arithmetic and boolean expression in Fig. 3 we can implement a

synthesizing function infer : list $(Var * Ty) \to Exp \to M\ (Ty * Exp')$ that returns a type and a type-reconstructed expression in an expression type Exp' that only allows type-annotated lets.

Consider for example the following code snippet that implements the let case:

$$\begin{aligned} &infer\ \Gamma\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = \\ &\quad (\tau_1, e_1') \leftarrow infer\ \Gamma\ e_1;\ (\tau_2, e_2') \leftarrow infer\ ((x, \tau_1) :: \Gamma)\ e_2;\ return\ (\tau_2, \textbf{let}\ x : \tau_1 = e_1'\ \textbf{in}\ e_2') \end{aligned}$$

After typechecking the first expression, we typecheck the second expression in an extended context, and finally produce the overall type and a type reconstructed let expression.

In our implementation, we defined the abstract monad interface using Coq's type class mechanism, and derived three instances: (1) for the option monad that eagerly solves constraints, (2) for a writer-transformed option monad that accumulates constraints, and (3) for a free monad. The last two implement phase separation.

Besides an interface for implementing TC, we also define a program logic interface (Fig. 2) for reasoning about such monadic computations in any of the three instances. We state reasoning rules for both *WP* and *WLP*. The return and bind rules are identical for *WP* and *WLP*. The interpretation of assert follows the traditional interpretation of guarded commands (see e.g. [4]). The last line of 2 states that weakest preconditions are monotonic for every computation. With the program logic in hand, we can establish correctness using statements similar to (1) and (2) from Sect. 2.2.

3.2 With unification

For existentially quantified types, we can introduce a non-deterministic operation *exists* that chooses an appropriate type. We extend the free monad and its program logics to support this operation:

Inductive $Free\ a := \ldots \mid Exists : (Ty \to Free\ a) \to Free\ a$

The result is that we modeled uvars using a shallow embedding by using variables of the meta language. Nevertheless, we can implement TC using this monad and prove it equivalent to a declarative specification. Unfortunately, it is not a monad that we can run.

To overcome this, we develop an alternative *deep embedding* of uvars. Similar to Keuchel et al. [3], we use possible world semantics to model the allocation of uvars, and implement TC in a Kripke-indexed monad, with worlds defined as sets of uvars and accessibility as set-inclusion. Using this alternative embedding we can implement a constraint generator that does not use higher-order features of the meta language in the representation.

Keuchel et al. [3] use a logical relation (LR) to reduce the soundness of a deep-embedding-based verification condition generator to a shallow one. We postulate, that the LR can be used in the TC setting to reduce the correctness of deeply-embedded to shallowly-embedded constraint generation.

4 RESULTS

My research is still ongoing and I have a partial mechanization in Coq of the results presented in this abstract. In particular, I have a finished proof of soundness and completeness of monadic TC for the arithmetic and boolean expression language described in Fig. 3. For TC with unification, I implemented TC for the simply-typed lambda calculus without type annotations in both the shallowly- and deeply-embedded constraint generation monads. For the shallow embedding, I also finished the soundness and completeness proofs, but the LR proof to connect deeply-embedded TC with shallowly-embedded TC is still missing. The solver for deeply embedded constraints, based on first-order unification by structural recursion [6], and its correctness proof was generously provided by the supervisor of my master thesis. The remainder is original work by the author.

REFERENCES

[1] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

[2] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[3] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified Symbolic Execution with Kripke-Specification Monads (and no Meta-Programming). (2022). Under submission..

[4] K. Rustan M. Leino. 2005. Efficient Weakest Preconditions. *Inf. Process. Lett.* 93, 6 (mar 2005), 281–288. https://doi.org/10.1016/j.ipl.2004.10.015

[5] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Mart'ınez, Cătălin Hriţcu, Exequiel Rivas, and 'Eric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (jul 2019), 29 pages. https://doi.org/10.1145/3341708

[6] Conor McBride. 2003. First-order unification by structural recursion. *Journal of functional programming* 13, 6 (2003), 1061–1075.

[7] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[8] James Hiram Morris. 1969. *Lambda-calculus models of programming languages.* Thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/64850

[9] François Pottier. 2014. Hindley-Milner Elaboration in Applicative Style: Functional Pearl. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 203–212. https://doi.org/10.1145/2628136.2628145

[10] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of functional programming* 24, 5 (2014), 529–607.

[11] Rafael Castro G. Silva, Cristiano Vasconcellos, and Karina Girardi Roggia. 2020. Monadic W in Coq. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity* (Natal, Brazil) *(SBLP '20)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3427081.3427085

[12] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28.

[13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. *ACM SIGPLAN Notices* 48, 6 (2013), 387–398.

[14] Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 440–451.

[15] P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[16] Mitchell Wand. 1987. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae* 10, 2 (1987), 115–121. https://doi.org/10.3233/FI-1987-10202