CAES9542 Technical English for Computer Science

Decidable Bounded Quantification

Partial Report

YANG Mingtian
(3035845576)

15th October 2024

1 Introduction

1.1 Overview

In the field of programming language theory, a type system is a formal system that categorises expressions into specific types in a compositional manner. A robust type system plays a crucial role in preventing certain undesirable program behaviours, thus enabling early detection of programming errors. By incorporating more features into their type systems, programming languages can achieve greater expressive capabilities. Nevertheless, the inclusion of new features into a type system may lead to increased metatheoretical complexity. Therefore, care must be taken when designing a type system.

Bounded quantification, which combines subtyping and parametric polymorphism, is a key feature in many modern programming languages, such as Java, Scala, and TypeScript. The concept of bounded quantification was initially introduced by Cardelli and Wegner [1985] in an experimental programming language called Fun. The type system of Fun was later refined and formalised into a calculus known as System $F_{<:}$ by Curien and Ghelli [1992].

In System $F_{<:}$, there are four kinds of types, namely the top type, variable types, function types, and universal types. The subtype relationship $<:$ is a preorder on them. The syntax for types can be described using Backus–Naur form as presented in Figure 1, and the static semantics for $<:$ can be depicted using the inference rules illustrated in Figure 2.

$$\tau ::= \top \mid \alpha \mid \tau \to \tau \mid \forall \alpha <: \tau . \tau$$

Figure 1: Syntax for types in System $F_{<:}$

$$\frac{}{\Gamma \vdash \sigma <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash \alpha <: \alpha}\ \textsc{Refl} \qquad \frac{\Gamma \vdash \Gamma(\alpha) <: \tau}{\Gamma \vdash \alpha <: \tau}\ \textsc{Var}$$

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}\ \textsc{Arrow}$$

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma, \alpha <: \tau_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall \alpha <: \sigma_1 . \sigma_2 <: \forall \alpha <: \tau_1 . \tau_2}\ \textsc{All}$$

Figure 2: Static semantics for $<:$ in System $F_{<:}$

While the rules are presented within a logical framework, they can also be effectively applied in a computational framework. From an algorithmic standpoint, these rules can be viewed as rules that reduce the main problem below the horizontal line into zero or more subproblems above the line. The subtyping algorithm recursively applies these rules and terminates once all subproblems have been resolved.

1.2 Problem Statement

While System $F_{<:}$ was widely adopted as the standard calculus for bounded quantification, it was eventually demonstrated to have an undecidable subtype relation [Pierce 1992]. In other words, the abovementioned subtyping algorithm does not terminate for all inputs, and it is impossible to design another subtyping algorithm for System $F_{<:}$ that guarantees its termination.

The source of the undecidability stems from the All rule, which changes the bound on instances of α in σ_2 from σ_1 to τ_1. Despite the subtype relationship between τ_1 and σ_1, τ_1 could potentially be structurally more complex than σ_1, leading to the subproblem becoming structurally larger than the initial problem.

1.3 Objectives and Deliverables

The main focus of this project is thus to investigate the design of a mechanism for bounded quantification that guarantees its decidability. This involves proposing new inference rules and using a interactive theorem prover to verify their desirable properties, such as reflexivity, transitivity, and decidability. Subsequently, a toy programming language incorporating the revised type system will be implemented. The ultimate goal is to improve the usability of programming languages and increase the productivity of programmers.

1.4 Outline of the Report

The remainder of this report proceeds as follows. In Section 2, prior research on decidable bounded quantification is reviewed, along with an examination of relevant techniques developed in recent works that may contribute to the achievement of decidable bounded quantification. Section 3 introduces the tools and methodology utilised and planned for this study.

2 Literature Review

Since the undecidability of System $F_{<:}$ became widely acknowledged, some decidable fragments of System $F_{<:}$ have been proposed and studied. A brief introduction of these type systems and their limitations is included in Section 2.1. Additionally, recent works have proposed new approaches for designing decidable type systems. The approach to be employed in this project is introduced in Section 2.2.

2.1 Decidable Fragments of System $F_{<:}$

2.1.1 Kernel $F_{<:}$

The undecidability of System $F_{<:}$ is attributed to the revision made by Curien and Ghelli [1992] to Fun's type system. In fact, the original formulation by Cardelli and Wegner [1985] already

gives rise to a decidable type system known as Kernel $F_{<:}$, where the subtype order is restricted to those with equal bounds. Its subtyping rule for universal types is as follows:

$$\frac{\Gamma, \alpha <: \sigma_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall \alpha <: \sigma_1 . \sigma_2 <: \forall \alpha <: \sigma_1 . \tau_2} \textsc{KernelAll}$$

The primary drawback of Kernel $F_{<:}$, which is also the reason for the revision by Curien and Ghelli [1992], is that the requirement for bounds to be identical significantly hampers the expressiveness of the programming language.

2.1.2 System $F^{\top}_{<:}$

Another decidable fragment of System $F_{<:}$ is System $F^{\top}_{<:}$ [Castagna and Pierce 1994], where α is rebounded to $\top$ during the inference on the bodies, leading to the disregard of type information regarding bounds. Its subtyping rule for universal types is as follows:

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma, \alpha <: \top \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall \alpha <: \sigma_1 . \sigma_2 <: \forall \alpha <: \tau_1 . \tau_2} \textsc{TopAll}$$

Although it ensures decidability and provides a more expressive subtype relation than Kernel $F_{<:}$, it does not interact well with the typing rules [Laird 2023], and thus is still not an ideal type system for bounded quantification.

2.1.3 Completely Bounded Quantification

Completely bounded quantification (CBQ) [Katiyar and Sankar 1992] can be viewed as a minimalistic decidable fragment of System $F_{<:}$. Instead of simply modifying the All rule, it imposes constraints on the formation of universal types. It prohibits $\top$ from appearing in the bounds to ensure that every type variable is completely bounded. As a result, if $\Gamma \vdash \tau_1 <: \sigma_1$ holds, the bounds τ_1 and σ_1 in the All rule must have the same structure, ensuring decidability. While CBQ is more restrictive and less expressive than Kernel $F_{<:}$, it serves as a suitable starting point for further research on the topic due to its simplicity.

2.2 Nominal Unfolding Rules

In a recent study by Zhou et al. [2023], nominal unfolding rules are employed to extend Kernel $F_{<:}$ with iso-recursive types and subtyping. The nominal unfolding rules are as follows:

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\hat{\alpha}}]A <: [\alpha \mapsto B^{\hat{\alpha}}]B}{\Gamma \vdash \mu\alpha.A <: \mu\alpha.B} \textsc{Nominal} \qquad \frac{\Gamma \vdash A <: B}{\Gamma \vdash A^{\hat{\alpha}} <: B^{\hat{\alpha}}} \textsc{Label}$$

In nominal unfolding rules, an additional label $\hat{\alpha}$ is introduced alongside the recursive variable name α, enabling the tracking of recursive variable names during double unfolding to avoid accidental subtyping.

The nominal unfolding rules have been formally proved by Zhou et al. [2022] to be sound, decidable, and equivalent in expressive power to the commonly utilised Amber rules for subtyping iso-recursive types. Additionally, Zhou et al. [2023] has illustrated that the nominal unfolding rules are more straightforward to be formalised and can be smoothly incorporated into an existing type system in a modular fashion.

Given these considerations, it is highly plausible that the concept of nominal labelling underlying the nominal unfolding rules represents a potent and suitable approach to achieving decidable bounded quantification.

3 Methodology

Before proposing the new type system, CBQ is first formalised. The tools and theories are elucidated in Section 3.1.

Building upon CBQ, a new type system featuring decidable bounded quantification will be proposed, drawing inspiration from the nominal unfolding rules presented in Section 2.2. The refined type system will be initially formalised using the same methodologies detailed in Section 3.1, followed by its implementation as a toy programming language using the technology stack outlined in Section 3.2.

3.1 Formalisation and Theorem Proving

Interactive theorem proving technology is employed to formalise CBQ. In contrast to traditional paper-and-pencil proofs, it ensures the accuracy of the proof via machine verification, provided that the theorem specifications are precise.

The theoretical basis for interactive theorem proving is grounded in computational trinitarianism, as illustrated in Figure 3.

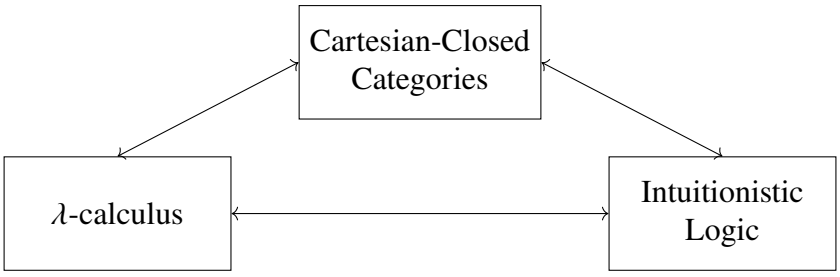

Figure 3: Computational trinitarianism [Melliès 2006]

In simple terms, logic, languages (illustrated by λ-calculus), and categories are three viewpoints of a unified computational concept. A proof of a proposition is computationally equivalent to a program of a specific type. Validating a proof essentially involves performing a type check on a program.

Specifically, Coq [The Coq Development Team 2024] is used as the proof assistant, and a Coq library called Metalib [Aydemir et al. 2008] is employed. This library offers practical infrastructures for formalising a type system in a innovative style that combines locally nameless representation and cofinite quantification in the definitions of term-wise relations [Aydemir et al. 2008]. This approach uses de Bruijn indices for bound variables and atoms for free variables to eliminate the need for α-conversion and the manipulation of shifting indices, thereby streamlining the proofs.

The proofs for reflexivity and transitivity are fairly standard and straightforward. The theorem of decidability, formalised with Coq as illustrated in Listing 1, is slightly more tricky to prove.

```
Theorem sub_decidability :
  forall E S T, wf_env E -> wf_typ E S -> wf_typ E T ->
  sub E S T \/ ~ sub E S T.
```

Listing 1: Theorem of decidability formalised with Coq

To prove decidability, a complexity metric developed by Katiyar and Sankar [1992] is employed:

$$
\begin{aligned}
\mathit{complexity}(\Gamma \vdash \sigma <: \tau) &= \mathit{size}(\sigma)_\Gamma + \mathit{size}(\tau)_\Gamma \\
\mathit{size}_\Gamma(\top) &= 1 \\
\mathit{size}_\Gamma(\alpha) &= \begin{cases} \mathit{size}_\Gamma(\Gamma(\alpha)) & \text{if } \Gamma(\alpha) \text{ is defined} \\ 1 & \text{otherwise} \end{cases} \\
\mathit{size}_\Gamma(\tau_1 \to \tau_2) &= \mathit{size}_\Gamma(\tau_1) + \mathit{size}_\Gamma(\tau_2) \\
\mathit{size}_\Gamma(\forall \alpha <: \tau_1 . \tau_2) &= \mathit{size}_\Gamma(\tau_1) + \mathit{size}_{\Gamma, \alpha <: \tau_1}(\tau_2)
\end{aligned}
$$

In prose, the complexity of a subtyping problem $\Gamma \vdash \sigma <: \tau$ is defined to be the sum of the textual size of σ and τ under the environment Γ.

With the complexity metric defined, the theorem `sub_decidability` can thus be proven using an auxiliary lemma, as shown in Listing 2.

```
Lemma sub_decidability_aux :
  forall k E S T, wf_env E -> wf_typ E S -> wf_typ E T ->
  complexity E S T < k ->
  sub E S T \/ ~ sub E S T.
```

Listing 2: Auxiliary lemma for proving `sub_decidability` formalised with Coq

The lemma `sub_decidability_aux` can be proven by inducting on k. The formalised proof demonstrates that $\mathit{complexity}(\cdot)$ is both finite and positive for any finite subtyping problem, and

any subproblem generated by the inference rules always has a smaller size than the original problem. Therefore, the substyping algorithm for CBQ is decidable.

For the formalisation of the new type system to be proposed, similar techniques will be used, although the specific mathematical details may vary slightly.

3.2 Implementation

An interpreter, structured as depicted in Figure 4, will be developed to implement a toy programming language that integrates the refined type system.

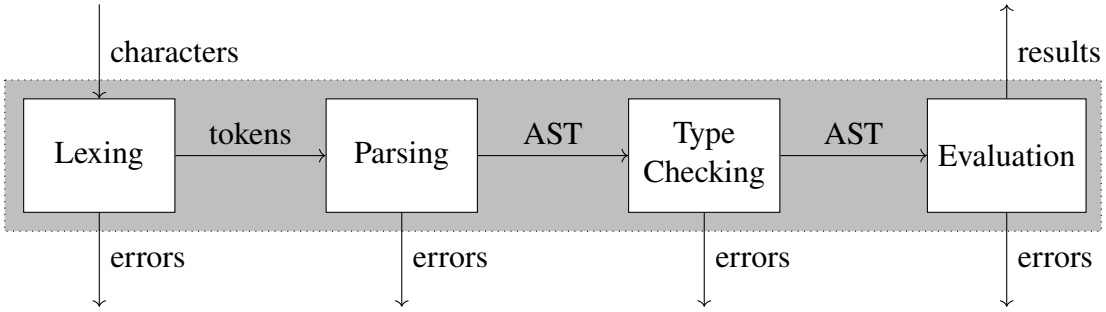

Figure 4: Structure of the interpreter

An interpreter is a program that takes source code input from the user and returns results or errors back to them. The interpreter workflow typically involves four phases: lexing, parsing, type checking, and evaluation.

During the lexing phase, the interpreter receives the source code as a stream of characters and converts it into a stream of tokens. In the parsing phase, the tokens are structured into an abstract syntax tree (AST). Subsequently, in the type checking phase, the interpreter verifies the validity of type information using the defined static semantic rules for the type system. Finally, in the evaluation phase, the code is executed on the AST by recursively applying the dynamic semantic rules, and the results are displayed to the user. Any errors detected at any phase are reported immediately.

In this project, OCaml [Leroy et al. 2024] will be used as the metalanguage throughout the four phases. OCaml's neat syntax for algebraic data types facilitates the manipulation of ASTs, enhancing development efficiency. For the first two phases, the OCaml libraries OCamllex and Menhir will be employed to automatically generate the lexer and parser, respectively, from the language specification. For the final two phases, both the type checker and the evaluator will be developed as functions that recursively apply the matched rule from a collection of rules.

References

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 3–15. https://doi.org/10.1145/1328438.1328443

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *Computing Surveys* 17, 4 (1985).

Giuseppe Castagna and Benjamin C. Pierce. 1994. Decidable bounded quantification. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/174675.177844

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science* 2, 1 (1992), 55–91.

Dinesh Katiyar and Sriram Sankar. 1992. Completely Bounded Quantification is Decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*. 68–77.

James Laird. 2023. Revisiting Decidable Bounded Quantification, via Dinaturality. *Electronic Notes in Theoretical Informatics and Computer Science* 1, Article 10 (Feb. 2023), 16 pages. https://doi.org/10.46298/entics.10474

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2024. The OCaml system release 5.2: Documentation and user's manual. (2024). https://ocaml.org/manual/5.2/

Paul-André Melliès. 2006. Functorial Boxes in String Diagrams. In *Computer Science Logic*, Zoltán Ésik (Ed.). Springer, Berlin, Germany, 1–30.

Benjamin C. Pierce. 1992. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 305–315. https://doi.org/10.1145/143165.143228

The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.2. https://coq.inria.fr/doc/V8.19.2/refman.

Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. 2023. Recursive Subtyping for All. *Proc. ACM Program. Lang.* 7, POPL, Article 48 (Jan. 2023), 30 pages. https://doi.org/10.1145/3571241

Yaoda Zhou, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2022. Revisiting Iso-Recursive Subtyping. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 24 (Sept. 2022), 54 pages. https://doi.org/10.1145/3549537