An Exceptional Type-Checker

Advancing Type-Checker Reliability with the Correct-by-Construction Approach for a Toy Language with Checked Exceptions

Supervisors: Jesper Cockx[1] and Sára Juhošová[1]

Mariusz Kicior[1] - m.a.kicior@student.tudelft.nl

1. INTRODUCTION

- Type-checkers allows programmers to specify and enforce constraints of types in values.
- Many type-checkers contain bugs [1] [2].
- One way of mitigating that, is following the Correct-by-Construction (CbC) approach.
- The CbC approach promotes early integration of specifications in program development.
- *How can the Correct-by-Construction approach be applied to develop a reliable type-checker for a programming language with checked exceptions?*

2. OBJECTIVES

- Develop a toy language with checked exceptions extension.
- Implement a type-checker following Correct-by-Construction approach.

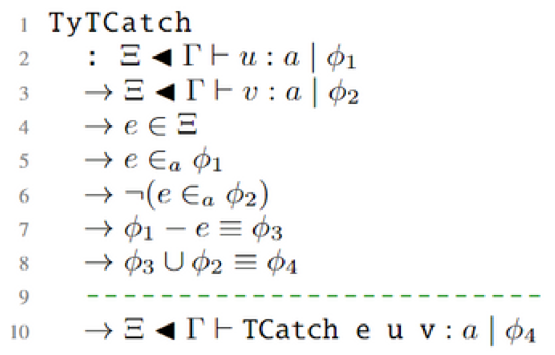
3. TOY LANGUAGE

- Toy language based on the Simply Typed Lambda Calculus with extensions for checked exceptions.
- No support for dynamic semantics.
- Terms: variables, lambda abstractions, applications, if-then-else statements, raising exceptions, catching exceptions, and declaring exceptions.
- Types: natural numbers, boolean values, and lambda expressions (functions).

4. TYPE AND EFFECT SYSTEM

$\boldsymbol{\Xi \blacktriangleleft \Gamma \vdash t : \tau \mid \phi}$

- **Ξ** - declared exceptions
- **Γ** - variables context
- **t** - the term being evaluated
- **τ** - the expected type of **t**
- **ϕ** - the set of exceptions that **t** could raise



```
TyTCatch
  : Ξ ◀ Γ ⊢ u : a | ϕ₁
  → Ξ ◀ Γ ⊢ v : a | ϕ₂
  → e ∈ Ξ
  → e ∈ₐ ϕ₁
  → ¬(e ∈ₐ ϕ₂)
  → ϕ₁ − e ≡ ϕ₃
  → ϕ₃ ∪ ϕ₂ ≡ ϕ₄
  ---------------------------------
  → Ξ ◀ Γ ⊢ TCatch e u v : a | ϕ₄
```

Figure 1. Typing rule for exception handling translated to Agda.

5. TYPE-CHECKER

- Bidirectional type-inference algorithm [3]: **synthesis** (*checkType*) and **inference** (*inferType*).
- Both functions return a typing judgement and exception annotations, with *inferType* also returning the type.
- Example type-checking routine (see Figure 2).

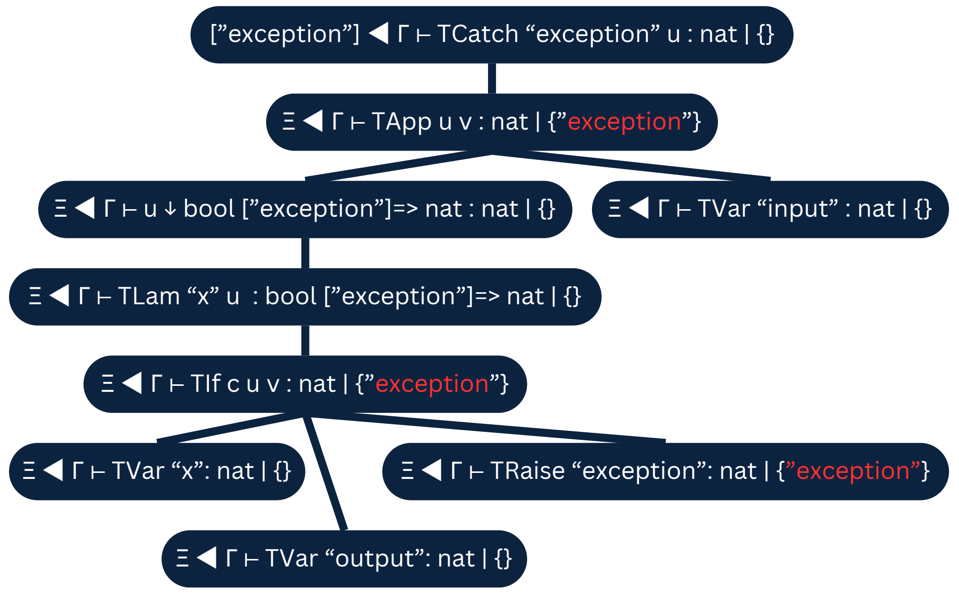

["exception"] ◀ Γ ⊢ TCatch "exception" u : nat | {}
Ξ ◀ Γ ⊢ TApp u v : nat | {"exception"}
Ξ ◀ Γ ⊢ u ↓ bool ["exception"]=> nat : nat | {}
Ξ ◀ Γ ⊢ TVar "input" : nat | {}
Ξ ◀ Γ ⊢ TLam "x" u : bool ["exception"]=> nat | {}
Ξ ◀ Γ ⊢ TIf c u v : nat | {"exception"}
Ξ ◀ Γ ⊢ TVar "x": nat | {}
Ξ ◀ Γ ⊢ TRaise "exception": nat | {"exception"}
Ξ ◀ Γ ⊢ TVar "output": nat | {}

Figure 2. AST of a type-checking algorithm.

6. CONCLUSION

The CbC approach is effective for developing reliable type-checkers, particularly when used with dependently typed languages like Agda. This method ensures that type rules are strictly enforced, providing strong guarantees about program correctness. Despite the steep learning curve, the benefits of this approach make it a viable strategy for similar projects in the future.

7. LIMITATIONS

- Lack of completeness: the produced algorithm is only sound.
- Language scope: we chose a minimalistic language but if extended, it could uncover even more insights.
- Generalization: we only treat exceptions while there's many other effects.

8. FUTURE WORK

- Completeness: make the type-checker complete.
- Testing: include testing as an additional validation of the produced algorithm.
- More effects: use this research as a starting point and expand it by adding more effects.

REFERENCES

[1] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. Type-centric kotlin compiler fuzzing: Preserving test program correctness by preserving types. 2020.
[2] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. 2022.
[3] Philip Wadler, Wen Kokke, and Jeremy G. Siek. Programming Language Foundations in Agda. 2022.

[1]EEMCS, Delft University of Technology, The Netherlands