An Exceptional Type-Checker

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

Mariusz Kicior[1]

Supervisor(s): Jesper Cockx[1]**, Sára Juhošová**[1]

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

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Mariusz Kicior
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Sára Juhošová, Thomas Durieux

Abstract

The Correct-by-Construction (CbC) programming paradigm has gained increasing attention, particularly with the rise of dependently typed languages. The CbC approach is often characterized as a rigid, rule-based construction process making it suitable for critical infrastructure like type-checkers, which are prone to bugs as they become more complex. However, the specific advantages and disadvantages of using the CbC approach for developing type-checkers in comparison to traditional programming languages remain unclear. Therefore, we investigate the development of a type-checker for a toy programming language extended with checked exceptions using the dependently typed programming language Agda. The results show that the CbC approach, combined with dependently typed languages, is highly effective for the very precise task of type-checker development. Despite the steep learning curve associated with these languages, this method offers notable benefits in ensuring the correctness and reliability of type-checkers. We conclude that this approach is a viable strategy for similar projects in the future.

1 Introduction

Type-checkers serve as critical tools for programmers, especially in the early stages of software development. They allow for the explicit specification and verification of program's types [7]. This not only helps in reducing straightforward errors but also plays a key role in maintaining code quality [1]. However, as the complexity of these tools increases, they become more prone to bugs. This can lead to the rejection of correct programs or the acceptance of incorrect ones.

Several studies on JVM compilers for widely used languages like Java, Kotlin, and Groovy highlighted the frequency of typing-related bugs. For instance, Chalios et al. identified 156 such bugs [2], followed by another study uncovering over 300 [3]. Remarkably, despite Chalios noting that the Scala repository labels "Typer" (type-checker related) issues as the most common bug type[1], it has remained unchanged. Furthermore, another study found additional 50 typing bugs in the Kotlin compiler [17]. Therefore, there is a need for mechanisms that can reduce the risk of such errors and further strengthen the programmer's confidence in the type-checker's accuracy.

These bugs may result from the traditional post-hoc verification programming approach, where a program is implemented first and verified later. One possible solution to address the mentioned issues in a type-checker implementation is the Correct-by-Construction (CbC) programming paradigm [13]. The CbC facilitates the integration of specification and program construction from the starting point of the process [16]. This methodology ensures that each development step adheres to a pre-defined specification, significantly minimizing the probability of errors as the program evolves.

[1] https://github.com/scala/bug/labels?sort=count-desc

One way of applying the CbC approach is by using dependently typed languages [13]. Such languages allow types to be parameterized by values. This means that the types can express richer properties of data, including invariants directly embedded in the type system. By embedding these constraints into the type system, dependently typed languages enable programmers to write specifications alongside the code that its type-checker can verify, which naturally leads to CbC software. Such software cannot compile unless it meets the specifications expressed in its types.

Using dependently typed languages to construct type-checkers is not a novelty. It has already been demonstrated by Pollack in 1995 [15] and has been recently revisited by Wadler et al. [18] using Agda. The latter work provides a comprehensive guide for building a type-checker, and this paper follows the methodology outlined in their book.

However, it remains unclear what the specific advantages and disadvantages are when using dependently typed languages compared to traditional programming languages. This study explores these differences, focusing on the practical implications of adopting dependently typed languages for type checking. To address this knowledge gap, we implement and analyze a type-checker for a toy programming language based on the simply typed lambda calculus (STLC) with the extension of checked exceptions. It highlights the challenges that arose during the process and discusses the benefits.

The remainder of this paper is structured as follows. Section 2 provides the necessary background knowledge to understand the research carried out in this project. Section 3 details the methodology and rationale behind the implementation of our type-checker, toy language, and typing rules. Section 4 elaborates on the actual implementation of the type-checker, its usage, and the insights gained from its development. Section 5 addresses responsible research concerns. Section 6 analyzes in detail the advantages and disadvantages of the CbC approach we encountered. Section 7 concludes the study and suggests directions for future work.

2 Background

This section provides the essential background information needed to understand our research in the following sections, and it explains why each term is relevant.

2.1 Simply Typed Lambda Calculus

Given our research goal to test the CbC approach in type-checker development, we choose to develop a type-checker for a simplified programming language based on the STLC, first introduced by Church [4]. It offers a concise but powerful framework for defining and manipulating functions. At its core are lambda terms, representing functions, and types, categorizing these functions based on their input and output types. Its simplicity makes it a perfect candidate for our type-checker implementation, offering a strong basis for the development of more advanced languages in the future.

2.2 Type Systems

The type system is the central element of the type-checker we are constructing. According to Pierce [14], "a type system is a tractable syntactic method for proving the absence of

certain program behaviors by classifying phrases according to the kinds of values they compute." This method allows for the detection of errors at compile time. It ensures that programs comply with specified constraints and prevent runtime errors (such as running wrong programs, which could lead to program crashes).

To understand type systems, it is essential to become familiar with the common notation used in their description. The notation used to describe type systems is expressed as $\Gamma \vdash e : \tau$. Here, Γ represents a context of variable typings, specifying the types of variables in a scope, the expression e refers to the program expression that is being typed, and τ denotes its type. The typing judgment $\Gamma \vdash e : \tau$ states that, given the assumptions specified in the context Γ, the expression e is of type τ. Also, the notation $\Gamma, x : \tau$ represents an extension of the context Γ with a new variable x bound to type τ.

Type systems have a crucial function in ensuring accuracy, security, and reliability in programming languages. They offer a framework for defining and enforcing rules on the manipulation of values. Furthermore, a type system guarantees that operations are executed on compatible types. Our type system is designed to handle a language similar to the STLC but with the addition of checked exceptions. This extension allows the type system to track not only the types of values but also the potential exceptions that functions might raise.

However, to build a type system for a language with checked exceptions, it is essential to extend our type system to account for effects. Effects model program behaviors that go beyond pure computation, such as exceptions, input/output operations, mutable state, and other side effects that interact with the external environment [10]. It is important to note that these effects are not types themselves but are tracked alongside types to ensure accurate modeling of program behavior.

2.3 Bidirectional Type Inference

Despite selecting a minimalistic toy language for our project, we choose to use a popular bidirectional type inference algorithm. This approach is known for its scalability compared to the Hindley-Milner algorithm and its widespread adoption in modern programming languages [6]. The algorithm is divided into two distinct phases: **checking** and **synthesis**. During the checking phase, the type-checker ensures that an expression corresponds to a given type. In the synthesis phase, it infers the type of an expression based on its context. This two-phase approach offers several advantages, including more predictability compared to other constraint-based inference algorithms (e.g., Hindley-Milner algorithm) and reduced reliance on explicit type annotations compared to languages without inference. It increases the overall usability of the type system, which is one of the reasons type inference is becoming standard even in statically typed languages like Scala [6].

2.4 Correct-by-Construction Programming

Traditionally, software is verified after its development, if it is verified at all. The CbC programming technique, which was popularized by Dijkstra [8], proposes an alternative paradigm. The CbC approach advocates for the early integration of specification in program construction. This methodology ensures that each development step aligns closely with predefined specifications, thereby reducing the likelihood of errors as the program evolves.

In the context of type-checkers, where correctness is of the highest importance, this approach becomes particularly relevant. By following the CbC approach, we should be able to greatly improve the reliability and trustworthiness of our type-checker. By doing so, we can ensure that the type-checker adheres rigorously to its typing rules throughout its development.

2.5 Agda

Agda is a total, dependently typed programming language. It is also used as a proof assistant and is characterized by its expressiveness and verification capabilities. It shares some similarities with Haskell in its functional programming approach, e.g., immutable data structures and treating functions as first-class citizens.

However, the distinctive feature of Agda is its support for dependent types, which allows datatypes types to be dependent on their values. This allows for embedding complex properties within type definitions. For example, a list of integers is not necessarily just typed as a list. It can also encode properties such as list length or order directly in its type, something that is not possible in Haskell.

```
raise : Term
raise = TRaise "Error: Division by zero."

declType : Type
declType = unit
```

Figure 1: Declaration of an exemplary expression and its type.

To illustrate the capabilities of dependent types in Agda, consider an example of testing a type-checker against a term using simplified syntax for clarity (see Figure 1). The `raise` variable represents a term that raises an exception for a "Division by zero" error, and its expected type `unit` is assigned to `declType`.

```
proof : checkType (Γ raise declType) ≡
    ↪ TyTRaise
proof = refl
```

Figure 2: Proof of type-checker correctness.

In order to guarantee the correctness of a type-checker using Agda's dependent types we embed and verify the properties directly within the type definitions in Figure 2. The `proof` function demonstrates how to validate that our type-checker correctly identifies the type of the `raise` term. The `proof` type specifies that the evaluation of the `raise` term by the `checkType` function within a given context should yield a result that matches `TyTRaise`. This `TyTRaise` represents the expected typing rule that our type-checker should return. The use of `refl` (reflexivity) in the value of `proof` asserts that both

sides of the equation are identical, providing a formal verification that the type-checker behaves as intended. (The syntax in this example is simplified and the actual functionality of our type-checker will be explained below.)

3 Methodology

In this section, we detail the methodology used to develop our bidirectional type-checker for a toy language with the extension of checked exceptions.

3.1 Toy Language

We design a toy language based on the STLC with extensions for checked exceptions. Since our goal is to create a type-checker and not a compiler, we focus on the language's syntax and typing rules, rather than its dynamic semantics. The language includes the following terms: variables, lambda abstractions, applications, if-then-else statements, raising exceptions, catching exceptions, and declaring exceptions. The language also supports natural numbers, boolean values, and lambda expressions (functions).

3.2 Typing Rules

The typing rules for our type-checker are presented in Figure 3. These rules define how terms in our toy language are typed. The notation has been updated from the common typing judgment notation previously introduced. We have extended it by introducing Ξ, which denotes the list of declared exceptions, and annotations ϕ, representing information about potential exceptions that terms may raise. Each rule is designed to derive the type of a term based on the types of its subterms and other contextual information, ensuring type safety even in the presence of exceptions. An arbitrary typing judgment $\Xi \blacktriangleleft \Gamma \vdash t : \tau \mid \phi$ should be read as follows: Ξ lists the declared exceptions, Γ provides the context, t is the term being evaluated with type τ, and ϕ identifies the exact set of exceptions that t could raise.

T-Var
This rule determines the type of a variable within a given context. If x is a variable in the context Γ, then the term TVar x has the type $\Gamma(x)$, under the set of declared exceptions Ξ and with no possible annotations represented by $\emptyset$.

T-Lam
This rule deals with lambda abstractions. If, under the extended context $(\Gamma, x : \tau_1)$ and exceptions Ξ, the term t has type τ_2 with annotations ϕ, then the lambda term TLam $x\,t$ is typed as $\tau_1[\phi] \Rightarrow \tau_2$ with annotation $\emptyset$; all the annotations are being 'redirected' to the lambda.

T-App
This rule handles function application. If t_1 is a function of type $\tau_1[\phi_1] \Rightarrow \tau_2$ with annotation ϕ_2, and t_2 is of type τ_1 with annotation ϕ_3, then the application of t_1 to t_2 results in a type τ_2 with annotations $\phi_1 \cup \phi_2 \cup \phi_3$.

T-Decl
This rule manages the declaration of exceptions. If the term t under the extended list of exceptions $(\{e\} \cup \Xi)$ has type τ with annotation ϕ, declaring the exception e for t maintains the type τ with the same annotation ϕ.

$$\frac{}{\Xi \blacktriangleleft \Gamma \vdash \text{TVar } x : \Gamma(x) \mid \emptyset}\ \text{[T-Var]}$$

$$\frac{\Xi \blacktriangleleft (\Gamma, x : \tau_1) \vdash t : \tau_2 \mid \phi}{\Xi \blacktriangleleft \Gamma \vdash \text{TLam } x\,t : \tau_1[\phi] \Rightarrow \tau_2 \mid \emptyset}\ \text{[T-Lam]}$$

$$\frac{\Xi \blacktriangleleft \Gamma \vdash t_1 : \tau_1[\phi_1] \Rightarrow \tau_2 \mid \phi_2 \quad \Xi \blacktriangleleft \Gamma \vdash t_2 : \tau_1 \mid \phi_3}{\Xi \blacktriangleleft \Gamma \vdash \text{TApp } t_1\,t_2 : \tau_2 \mid \phi_1 \cup \phi_2 \cup \phi_3}\ \text{[T-App]}$$

$$\frac{\{e\} \cup \Xi \blacktriangleleft \Gamma \vdash t : \tau \mid \phi}{\Xi \blacktriangleleft \Gamma \vdash \text{TDecl } e\,t : \tau \mid \phi}\ \text{[T-Decl]}$$

$$\frac{e \in \Xi}{\Xi \blacktriangleleft \Gamma \vdash \text{TRaise } e : \tau \mid \{e\}}\ \text{[T-Raise]}$$

$$\frac{\Xi \blacktriangleleft \Gamma \vdash t_1 : \tau \mid \phi_1 \quad \Xi \blacktriangleleft \Gamma \vdash t_2 : \tau \mid \phi_2}{\Xi \blacktriangleleft \Gamma \vdash \text{TCatch } e\,t_1\,t_2 : \tau \mid (\phi_1 - \{e\}) \cup \phi_2}\ \text{[T-Catch]}$$

$$\frac{\Xi \blacktriangleleft \Gamma \vdash t_1 : \text{bool} \mid \phi_1 \quad \Xi \blacktriangleleft \Gamma \vdash t_2 : \tau \mid \phi_2 \quad \Xi \blacktriangleleft \Gamma \vdash t_3 : \tau \mid \phi_3}{\Xi \blacktriangleleft \Gamma \vdash \text{TIfThenElse } t_1\,t_2\,t_3 : \tau \mid \phi_1 \cup \phi_2 \cup \phi_3}\ \text{[T-If]}$$

$$\frac{\Xi \blacktriangleleft \Gamma \vdash u : \tau \mid \phi}{\Xi \blacktriangleleft \Gamma \vdash (u \downarrow \tau) : \tau \mid \phi}\ \text{[T-Ann]}$$

Figure 3: Type rules

T-Raise

This rule addresses the raising of exceptions. If e is an exception within the declared set Ξ, then the term TRaise e is typed as τ with annotated by a singleton set $\{e\}$.

T-Catch

This rule covers exception handling. If t_1 could raise exception e and is typed as τ with annotation ϕ_1, and t_2 has type τ with a different annotation ϕ_2, then the term TCatch $e\,t_1\,t_2$ is typed as τ with annotations $(\phi_1 - \{e\}) \cup \phi_2$, which basically means TCatch catches a single exception and removes it from the annotations.

T-If

This rule applies to conditional expressions. If the condition t_1 is typed as boolean with annotation ϕ_1, and both branches t_2 and t_3 have type τ with annotations ϕ_2 and ϕ_3 respectively, then the term TIfThenElse $t_1\,t_2\,t_3$ results in type τ with annotations $\phi_1 \cup \phi_2 \cup \phi_3$.

T-Ann

This rule is necessary for our type system implementation to explicitly type annotated lambda terms. This approach splits type checking into checking and inference, simplifying implementation but limiting type inference for lambda terms without annotations. Following the PLFA approach [18], the `T-Ann` rule allows explicit term annotations.

3.3 Type-checker Design

Our approach focuses on producing a sound, instead of a complete, algorithm. This means, when the type-checker checks a correctly typed expression it returns proof of that. Whereas when it encounters a type error, it throws an informative message, rather than a proof of incorrectness. We chose to focus on soundness because it already reveals the main advantages and challenges of our approach, and in a compiler, detailed error message is more useful than proof of incorrectness.

We begin with a basic implementation that includes only the fundamental types of the STLC: variables, abstractions, and applications. These terms are paired with base types for natural numbers, boolean values, and function types. To make our implementation capable of handling more real-world programs, we extend it by adding explicit type annotations and if-then-else terms. Following this, we further extend our type system to support checked exceptions. This extension requires incorporating effects into our type-checker, as exceptions cannot be treated as types without violating the principle that each term has a single type in the STLC [18].

To manage the addition of effects, we adopt the approach outlined by Mogi [11] and further elaborated by Daan [9]. This approach separates values and computations, assigning distinct types to each. In practical terms, this means that effects can only occur in the context of function types, not in other values (e.g., a variable cannot raise an exception).

We introduce new terms to handle exceptions: declare exception, raise exception, and catch exception, and we update the existing type rules to integrate these new terms in the same fashion as in our typing rules. We achieve it by creating two new 'contexts', one for all declared exceptions (the user has to declare an exception before using it) and another for the annotations. The annotations are used to track which exceptions a term might raise.

4 Type-checker Implementation

In this section, we explain our type-checker implementation, the choices we make, and compare some aspects with how they could be implemented in Python, serving as an arbitrary (not dependently typed) programming language. This comparison showcases the challenges we face and how they could be handled in an unverified context.

4.1 Context Implementation

A type system requires a context to store information about variables, functioning similarly to a dictionary-like data structure. However, Agda's standard library lacks an implementation for that. Thus, we introduce our own `Context` type alias (see Figure 4). It is indexed by Scope, which in our implementation is a list of all names introduced in the given context. This way, we enforce and track the relation between variable names and their corresponding types within a type system.

```
Context : (v : Set) → (α : Scope) → Set
Context v α = All (λ _ → v) α
```

Figure 4: Definition of the `Context` type alias in Agda, indexed by `Scope`, which tracks variable names and their corresponding types.

We also add functionality for extending the context using the operator `_,_:_`, which adds a new name to the scope and assigns it a type (see Figure 5). Additionally, we implement a function `lookupVar` that allows for looking up a variable in the context and returns its type. This function requires proof of membership, ensuring that we never attempt to look up a variable that is not in the context.

```
Context : (v : Set) → (α : Scope) → Set
Context v α = All (λ _ → v) α

_,_:_ : Context v α → (x : name) → v →
    ↪ Context v (x :: α)
_,_:_ ctx _ v = v :: ctx

lookupVar : (Γ : Context v α) (x : name) (
    ↪ proof : x ∈ α) → v
lookupVar (v :: _ ) x here = v
lookupVar (_ :: ctx) x (there proof) =
    ↪ lookupVar ctx x proof
```

Figure 5: Functions for extending the context and looking up variable types in Agda.

Figure 6 presents an alternative implementation of context management in Python found in an open-source repository,[2] where it is implemented as a simple dictionary.

```
class Context(dict):
  def forkwith(self, variable, ttype) ->
    ↪ Context:
      context_response = self.get(variable)
      if context_response is not None and
    ↪ context_response != ttype:
          raise ContextCorruption()
      new_context = deepcopy(self)
      new_context[variable] = ttype
      return new_context
```

Figure 6: Context management in Python.

4.2 Annotation Implementation

The type-checker also requires a set-like datastructure. We choose to create a new datatype, `Ann`, to provide more ex-

[2] https://github.com/magniff/types101

pressiveness and functionality similar to set operations, such as union and difference. This approach is more compatible with the specifications of our typing rules in comparison to using a simple list of strings. The Ann datatype consists of two constructors: $\emptyset$, representing the empty set, and _+++_, which adds a string annotation to an existing set (see Figure 7). This structure enables us to rigorously and type-safely define operations such as union and membership.

```
data Ann : Set where
  ∅ : Ann
  _+++_ : Ann → String → Ann
```

Figure 7: Definition of the Ann datatype in Agda.

As our typing rules suggest, we need a union operation. It is implemented as the _∪_≡_ datatype, which ensures that the union of two annotation sets is correctly defined and produces a new set (see Figure 8). The empty and append constructors of this datatype provide the union rules: empty states that the union of any set with the empty set is the set itself, while append handles the addition of a new annotation to the union of two sets.

```
data _∪_≡_ : Ann → Ann → Ann → Set where
  empty : ∀ {ϕ} → ϕ ∪ ∅ ≡ ϕ
  append : ∀ {ϕ₁ ϕ₂ ϕ₃ v} → ϕ₁ ∪ ϕ₂ ≡ ϕ₃ →
    ↪ ϕ₁ ∪ (ϕ₂ +++ v) ≡ (ϕ₃ +++ v)
```

Figure 8: Definition of the _∪_≡_ datatype in Agda, specifying the rules for the union operation of annotation sets with empty and append constructors.

The un function performs the union of two annotation sets and returns the resulting set along with proof that the union operation is performed correctly (see Figure 9). This function uses pattern matching and recursion to traverse and combine the annotation sets.

```
un : (ϕ₁ ϕ₂ : Ann) → ϕ₃ ∈ Ann , ϕ₁ ∪ ϕ₂ ≡ ϕ₃
un ϕ₁ ∅ = ϕ₁ , empty
un ϕ₁ (ϕ₂ +++ x) with un ϕ₁ ϕ₂
... | ϕ₃ , ϕ₃-proof = (ϕ₃ +++ x) , append ϕ₃-
    ↪ proof
```

Figure 9: Implementation of the un function in Agda.

Membership in an annotation set is defined by the _∈_ datatype, which includes the constructors here and there (see Figure 10). The here constructor indicates that an annotation is present in the set, while there allows us to traverse the set to check for the presence of an annotation. We also introduce the _∈?_ function, which serves as a membership proof generator (see Figure 10). This function automatically determines if an annotation is present in the set, eliminating the need for a user-written proof. Similarly, we implemented a function for computing the difference between annotation sets.

```
data _∈_ : String → Ann → Set where
  here : ∀ {v ϕ} → v ∈ (ϕ +++ v)
  there : ∀ {v w ϕ} → v ∈ ϕ → v ∈ (ϕ +++ w)

_∈?_ : (x : String) (set : Ann) → Dec (x ∈
    ↪ set)
x ∈? ∅ = no (λ ())
x ∈? (a +++ ϕ) with x ≡ a
... | yes refl = yes here
... | no x≢a with x ∈? ϕ
...     | yes p = yes (there p)
...     | no n = no (λ { here → contradiction
    ↪  refl x≢a ; there p' → n p })
```

Figure 10: Definitions for checking membership in an annotation set in Agda.

In contrast, Python provides built-in support for set operations, which could greatly simplify the implementation. Figure 11 shows a simple example of managing annotations using Python's set data structure. Here, we create an annotation set, add elements to it, and perform a union operation with another set within a newly created object Ann.

```
class Ann:
  def __init__(self):
      self.annotations = set()
  def add_annotation(self, annotation: str):
      self.annotations.add(annotation)
  def union(self, other: Ann) -> Ann:
      result = Ann()
      result.annotations = self.annotations.
    ↪ union(other.annotations)
      return result
  def __contains__(self, annotation: str) ->
    ↪ bool:
      return annotation in self.annotations
```

Figure 11: Hypothetical annotation management in Python.

4.3 Typing Rules in Agda

Next, we focus on translating the typing rules to Agda, specifically those concerning exceptions, as these are the main focus of our work. We create a new datatype for these rules with the following signature: data _◀_⊢_:_|_ (Ξ : ↪ List String) (Γ : Context Type α) : Term α → ↪ Type → Ann → Set. This aligns precisely with the extended notation of typing judgments presented above. Each typing rule is basically a constructor of this datatype. We begin with the explanation of a lambda typing rule.

```
TyTLam
  : Ξ ◀ (Γ, x : a) ⊢ u : b | ϕ₁
  ----------------------------------------
  → Ξ ◀ Γ ⊢ TLam x u : a [ϕ₁] ⇒ b | ∅
```

Figure 12: Typing rule for lambda terms.

The `TyTLam` constructor (see Figure 12) adheres to the behavior specified in the original typing rule. It captures all exceptions that may be thrown within its body and assigns them as lambda annotations. This allows the lambda to 'consume' these exceptions. In other words, the lambda type $a[\phi_1] \Rightarrow b$ encapsulates all the annotations raised within its body. These annotations are managed by the lambda until the lambda term is applied using the `TyTApp` constructor.

```
TyTApp
  : Ξ ◀ Γ ⊢ u : a  [ϕ₁] ⇒ b | ϕ₂
  → Ξ ◀ Γ ⊢ v : a | ϕ₃
  → ϕ₁ ∪ ϕ₂ ≡ ϕ₄
  → ϕ₃ ∪ ϕ₄ ≡ ϕ₅
  ----------------------------------------
  → Ξ ◀ Γ ⊢ TApp u v : b | ϕ₅
```

Figure 13: Typing rule for application.

The `TyTApp` constructor (see Figure 13) handles the application of functions. It takes a term of lambda type, ensures its argument is of type `a`, and performs the sum operation using the `_∪_≡_` datatype declared for the `Ann` datatype. This operation 'activates' the annotations carried by the lambda expression by adding them to the annotation set again, which means the annotated exceptions can be raised by the application term. Now, that we know how these lambda annotations can return to the program, we move on to catching them.

```
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 14: Typing rule for exception handling.

The `TyTCatch` constructor (see Figure 14) introduces two branches. The first branch is annotated by a set of exceptions ϕ_1 containing an exception `e`, and the second branch specifically does not contain this exception. The final annotation of this rule consists of the union of the two sets ϕ_1 and ϕ_2 with the exception `e` removed. This ensures that the annotation is not reintroduced into the program.

4.4 Bidirectional Type Inference Algorithm

Once we translate terms, types, and typing rules to Agda, we can implement the type inference algorithm. The `Evaluator` monad serves as a wrapper that helps us build a sound algorithm. It is defined generically to either return a type `a` if the computation succeeds or a `String` representing an error via the `EvalError` function. This setup allows our type inference algorithm to handle errors gracefully. An `Evaluator` computation results in either a successful outcome of type `a`, which in our case is a typing judgment `_◀_⊢_:_|_`, or a string describing the error.

With this foundation, we can proceed to define the type inference algorithm. For brevity, we will not explain each type checking rule but will describe the general approach. The type inference procedure comprises two mutually recursive functions: inference and synthesis. These functions are translated to `inferType` and `checkType`, respectively, whose signatures are shown in Figure 15.

```
inferType : ∀ (Ξ : List String) (Γ : Context
    ↪ Type α) u → Evaluator (Σ[ t ∈ Type ]
    ↪ Σ[ ϕ ∈ Ann ] Ξ ◀ Γ ⊢ u : t | ϕ)
checkType : ∀ (Ξ : List String) (Γ : Context
    ↪ Type α) u (ty : Type) → Evaluator (Σ[
    ↪  ϕ ∈ Ann ] Ξ ◀ Γ ⊢ u : ty | ϕ)
```

Figure 15: Signatures of `inferType` and `checkType` functions.

inferType

The `inferType` function attempts to infer the type of a term `u` within a given context Γ and environment Ξ, resulting in an `Evaluator` computation that produces a type `t` and annotations ϕ. The `checkType` function, on the other hand, checks whether a term `u` conforms to a specified type `ty` within the same context and environment, resulting in an `Evaluator` computation that produces the annotations ϕ.

```
inferType Ξ Γ (TApp lam arg) = do
  -- Infer types for the head and argument of
    ↪  the application.
  (a [ ϕ₁ ]⇒ b , (ϕ₂ , tr₁)) ← inferType Ξ Γ
    ↪  lam
    where _ → evalError "Application head
    ↪ should have a function type!"
  (ϕ₃ , tr₂) ← checkType Ξ Γ arg a
  -- Sum the annotations of the head and
    ↪ argument.
  let (ϕ₄ , ϕ₄-proof) = un ϕ₁ ϕ₂
      (ϕ₅ , ϕ₅-proof) = un ϕ₃ ϕ₄
  return (b , (ϕ₅ , TyTApp tr₁ tr₂ ϕ₄-proof ϕ₅
    ↪ -proof))
```

Figure 16: Implementation of the `inferType` function for application.

The structure of `inferType` involves matching on the term `u` and handling various cases. For instance, variable terms return their type from the context, while lambda terms require explicit type annotations. For applications, as illustrated in Figure 16, `inferType` infers the types of both the function (`lam`) and its argument (`arg`), combines their annotations, and returns the result type. Specifically, the function's type and annotation are inferred first, ensuring it has a function type, and then the argument's type is checked against this inferred type. The annotations are summed to maintain consistency. Annotated terms are checked against their provided types, ensuring the overall consistency of the inferred types.

checkType

The `checkType` function is responsible for ensuring that a term `u` matches a specified type `ty`. It handles various constructs by recursively invoking `checkType` and `inferType` as needed. For example, when handling an exception, as shown in Figure 17, `checkType` first verifies if the exception `e` is declared within the context Ξ using Agda's standard library membership proof generator `_∈?_` for a String in a list. If an exception is not declared, it raises an error. Otherwise, it checks the types of both the exception term and the handler term against the specified type `ty`. It then calculates the subtraction and sum of annotations to ensure consistency. The function ensures that the exception term is annotated correctly and that the handler term does not carry the exception annotation, thereby maintaining type correctness in the presence of exceptions.

```
checkType Ξ Γ (TCatch e exceptionTerm
    ↪ handleTerm) ty with e ∈? Ξ
... | no _ = evalError "Catching a not
    ↪ declared exception!"
... | yes (e∈Ξ) = do
  -- Check types for the exception term and
    ↪ the exception handler.
  (ϕ₁ , tr₁) ← checkType Ξ Γ exceptionTerm
    ↪ ty
  (ϕ₂ , tr₂) ← checkType Ξ Γ handleTerm ty
  -- Calculate the subtraction.
  let (ϕ₃ , ϕ₃-proof) = removeAnn ϕ₁ e
  -- Calculate the sum of terms.
  let (ϕ₄ , ϕ₄-proof) = un ϕ₃ ϕ₂
  (yes , e∈ϕ₁) ← e ∈? ϕ₁
    where _ → evalError "Exception term
    ↪ should be annotated!"
  (no , e∉ϕ₂) ← e ∈? ϕ₂
    where _ → evalError "Exception handler
    ↪ should not be annotated with the
    ↪ exception!"
  return (ϕ₄ , TyTCatch tr₁ tr₂ e∈Ξ e∈ϕ₁ e∉ϕ₂
    ↪  ϕ₃-proof ϕ₄-proof)
```

Figure 17: Implementation of the `checkType` function for exception handling.

5 Responsible Research

The research presented in this paper involves logic and mathematical methods, and so avoids ethical concerns typically linked to data-driven studies, such as violations of privacy or misuse of data. The reported conclusions are directly derived from internally developed code, eliminating the use of any external data, unless specified otherwise. In order to address the reproducibility issues, the complete codebase is made available to the public.[3]

The theoretical framework presented in this study should provide readers with the necessary understanding to comprehend the code stored in the repository. Additionally, it offers an explanation of bidirectional type inference and a basic understanding of Agda. For further details on the Agda language, the "Programming Language Foundations in Agda" book [18] is recommended.

[3]https://github.com/kmariuszk/cbc-type-checker

6 Discussion

The CbC approach has been a subject of multiple studies in formal methods in prior research [5]. However, despite its potential, the CbC paradigm has not been widely adopted, mainly due to the significant resources and complexity involved in deploying it at scale. In Section 4, our objective was to demonstrate these problems by presenting Python code alternatives that were far more concise and easier to understand in comparison to the same code in Agda. Nevertheless, our project showcases that it is indeed feasible to implement an effective type-checker using this technique. Designing a new programming language is inherently a deliberate task, and the creation of typing rules is an unavoidable part of it.

Once the typing rules are formulated, they can be converted to Agda. Its support for Unicode allowed us to use mathematical syntax directly in our implementation, which reduced the cognitive load during the translation of rules into code and when reasoning about it. Additionally, Agda's standard library facilitates the automatic generation of certain constraints. For instance, we utilized list membership proof generation for exception-specific rules. By pattern-matching our code we could verify whether an exception was part of the context of the declared exceptions as presented in Figure 17.

However, the development process was not without challenges. Since we wanted to benefit from Agda's standard library's functionality we needed to limit our exceptions to be only Strings. This simplification was necessary to continue using the functionality from the `Setoid` library. Otherwise, we would need to provide the `Annotation` module with an equality property which turned out to be impractical and problematic given our expertise in Agda. The need for providing this equality property is a challenge common to both dependently typed and regular languages. However, the multiple layers of abstraction over a simple equality concept became difficult to navigate, worsened by the lack of tutorial resources on working with setoids.

Additionally, we had to implement set and map-like data structures ourselves as described in Section 4. Initially, our set implementation faced issues where, despite Agda's code compiling, it did not function properly. This was because Agda's unification algorithm could not determine whether the union of lists of strings we proved to be equal was indeed equal. This issue presents the challenge of working with Agda, requiring a deeper understanding and careful handling of type equivalence.

Furthermore, although Agda allows for proof of code correctness, it requires very precise typing rules, which themselves may be flawed. As a simple example, Agda will not automatically identify contradictory rules such as one presented in Figure 18. Even though the given example is trivial, it showcases that the responsibility for defining accurate typing rules lies entirely with the developer. Consequently, the CbC approach is only as reliable as the rules you define and

Agda will not assist in identifying flaws in those rules.

```
TyTRaise
    : e ∈ Ξ
    → e ∉ Ξ
    ----------------------------
    → Ξ ◀ Γ ⊢ TRaise e : a | ϕ
```

Figure 18: Example of a contradictory typing rule translated to Agda, demonstrating a flaw that Agda cannot automatically detect.

Employing the CbC approach to implement type-checkers turned out to be a valid approach. While it does impose more challenges from the developing perspective, the correctness of the produced program and the ease of transitioning between the theoretical definition of a language and its representation in Agda using these languages make it a viable approach. After a programmer invests significant effort into ensuring their code adheres to the given set of rules, they can be confident that it will work as intended once compiled.

7 Conclusion and Future Work

Based on the findings presented in this paper, we conclude that the CbC approach, combined with Agda, is suitable for developing type-checkers. One of the main benefits of using Agda in this context is its strict enforcement of typing rules. Once these rules are established, they cannot be violated, which guarantees consistency and reliability of the implementation. This approach eliminates false positive results, meaning no ill-typed program will be accepted by our type-checker. However, this strictness also presents a challenge, particularly in the complexity of writing proofs. There are numerous potential extensions to our type-checker that could further explore and validate the CbC approach.

One of the most significant extensions would be to develop a program that is not only sound but also complete. We suggest following the methodology presented in Philip Wadler's book [18], which includes implementing proofs of incorrectness for wrongly typed expressions. Even though we choose to develop a sound algorithm that generates meaningful error warnings, extending this to a complete algorithm could uncover even more insights.

In addition, the CbC approach could be augmented with a comprehensive test suite. This could be achieved by translating our Agda code to Haskell using `Agda2Hs`[4] tool. Once translated to Haskell, the QuickCheck library[5] could be used to perform property-based testing. This will improve the reliability of the program and ensure that the typing rules are correct. Such a test suite could be built by following work of Pałka et al. [12], who demonstrated the possibilities of QuickCheck in identifying bugs through random testing.

Additionally, it is important to highlight that the typing rules used in the type-checker development did not produce any bugs during manual testing. However, we have not formally proven their soundness and preservation. These proofs are a necessary addition to the developing process to ensure that the set of typing rules itself is both correct and sound.

In summary, although our current implementation showcases the possibility and benefits of using the CbC technique for developing type-checkers, there is still room for additional research and improvement. Future studies should focus on extending the type-checker to reach completeness and integrating a testing framework. This would expand our understanding of the applicability of the CbC methodology in type-checker development.

[4]https://github.com/agda/agda2hs

[5]https://hackage.haskell.org/package/QuickCheck

Acknowledgments

I would like to thank my peers who worked on the same research project for their support throughout. I am grateful to Professor Jesper Cockx and my supervisor Sára Juhošová for their guidance. I appreciate the assistance of Cas van der Rest in developing the typing rules for exceptions.

References

[1] Patrick Amey. Correctness by construction: Better can also be cheaper. *CROSSTALK, The Journal of Defense Software Engineering*, 15, 2002.

[2] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. Well-typed programs can go wrong: a study of typing-related bugs in jvm compilers. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021.

[3] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. page 183–198, 2022.

[4] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[5] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[6] Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. 2020.

[7] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. pages 758–769, 2017.

[8] Derrick Kourie and Bruce Watson. *The Correctness-by-Construction Approach to Programming*. 2012.

[9] Daan Leijen. Type directed compilation of row-typed algebraic effects. page 486–499, 2017.

[10] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. page 47–57, 1988.

[11] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[12] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, page 91–97, New York, NY, USA, 2011. Association for Computing Machinery.

[13] Alberto Pardo, Emmanuel Gunther, Miguel Pagano, and Marcos Viera. An internalist approach to correct-by-construction compilers. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, New York, NY, USA, 2018. Association for Computing Machinery.

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

[15] Robert Pollack. A verified typechecker. pages 365–380, 1995.

[16] T. Runge, T. Bordis, A. Potanin, T. Thüm, and I. Schaefer. Flexible correct-by-construction programming. 2022.

[17] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. Type-centric kotlin compiler fuzzing: Preserving test program correctness by preserving types. 2020.

[18] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. 2022.