Correct-by-Construction Type-Checking for Algebraic Data Types

Implementing a Type-Checker in Agda

Miloš Ristić[1]

Supervisors: 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: Miloš Ristić
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Sára Juhošová, Thomas Durieux

Abstract

Type-checkers are used to verify certain attributes of programs are correct. Avoiding bugs in type-checkers is especially important when accepting faulty programs has serious real-world consequences. Correct-by-construction programming aims to prevent such bugs by embedding proofs of a program's correctness within the program itself. However, this approach introduces different challenges, such as added complexity. Whether the benefits of correct-by-construction type-checkers outweigh these challenges for more complex language features remains uncertain. This paper investigates correct-by-construction type-checking for a toy language with polymorphic algebraic data types and pattern matching. We do this by implementing a type-checker in the dependently typed language Agda. We show that this approach guarantees a type-checker that does not accept ill-typed terms. Furthermore, we reflect on the challenges of this approach and argue that this approach should be used when a guarantee of correctness is required.

1 Introduction

Static type systems are a form of (lightweight) formal verification for software systems [1]. They allow the programmer to specify their intention regarding computation, which is then validated at compile time. One example is annotating a function's argument with its expected type, preventing incorrect function application, e.g., integer multiplication with a boolean. More advanced examples include preventing out-of-bound indexing using dependent types or restricting resource use using substructural types [2]. Static type systems can prevent a wide range of software bugs. Because of this, many popular programming languages use them, which shows their significance in software engineering.

Type-checkers used for validating type systems can contain bugs themselves. Ideally, type-checkers are complete and sound, meaning they do not incorrectly reject a correct program and always reject ill-typed programs [3]. While incompleteness merely annoys the developer, unsoundness could lead to critical system failure. Avoiding bugs in type-checkers is important, especially for fields where accepting a faulty program can have serious real-world consequences, e.g., military or aerospace.

One area aiming to solve these problems is correct-by-construction (CbC) programming. This is a style of programming that allows intrinsic verification. Intrinsic verification is obtained by defining properties the program must adhere to at the type level [4], and then writing a program that satisfies these properties by construction. If the program compiles, it is correct for the properties that were defined. Encoding invariants in types is a well-known feature of dependent types [5], meaning languages with this feature, e.g. Agda, are well-suited for this line of research.

There are existing resources on CbC programming with type systems. Wadler et al. [3] use this style of programming to formalize the simply-typed lambda calculus (STLC), and they extend it with further features such as let constructs and sum types. Casamento [5] presents a CbC type-checker for a language based on the STLC with a module system using scope graphs. Sozeau et al. [6] implement a CbC type-checker for the proof assistant Coq in Coq. They highlight the importance of the CbC approach, noting that every year at least one critical bug is found in the implementation of Coq.

It remains uncertain whether the benefits CbC programming introduces outweigh the challenges for more complex language features. One feature worth exploring is algebraic data types (ADTs), which expand expressive power by allowing programmers to declare and use common data types such as lists, trees, and more. While Coq supports inductive types which can be used as ADTs, Sozeau et al. focus on Coq as a whole, whereas we focus on ADTs and the challenges we encounter for them specifically.

This paper answers the following research question: *How can correct-by-construction programming be used to increase the trustworthiness of type-checkers for algebraic data types?* To do this, we first introduce a toy language involving ADTs, pattern matching, and polymorphism. After this, we translate the syntax and typing rules into Agda, which serves as the meta language. Then, we explain the implementation of the CbC type-checker. Following that, we reflect on the value of the CbC approach and discuss any challenges we encountered.

We present the following contributions:

1. The syntax and typing rules of a toy language with ADTs, pattern matching, and polymorphism.
2. An implementation for a CbC type-checker in Agda written for the toy language.
3. A discussion on the benefits and challenges the CbC approach brings.

The paper begins by giving some relevant background information in Section 2. Section 3 introduces the toy language used for the type-checker. Section 4 outlines the implementation of the CbC type-checker. Section 5 discusses the type-checker and reflects on the CbC approach. In Section 6, we reflect on the ethical aspects of our research. Section 7 presents the conclusions and suggestions for future work. Finally, Section 8 marks the end of this paper with the acknowledgments.

2 Background

This section briefly discusses some background knowledge that is required to understand the further sections. We first discuss type systems in Section 2.1. After that, we go over what ADTs are in Section 2.2. Finally, we discuss Agda in Section 2.3.

2.1 Type Systems

Type systems dictate what type each value within a programming language has and how these types can interact. They can enforce rules statically (at compile-time), dynamically (at run-time), or using a combination of both. Statically typed languages are checked using a type-checker. While static checks mainly serve to prevent execution errors at run-time [7], they can also increase the efficiency of a program by eliminating

many of the dynamic checks that would otherwise have to be performed [1; 7].

Type inference is when the compiler automatically deduces the types of certain terms based on the context surrounding the term. This has the advantage that explicit annotations can be omitted while still ensuring type safety. Excluding explicit annotations makes the code more concise and removes the programmer's burden of specifying types manually.

2.2 Algebraic Data Types

ADTs are types composed of other types. This can be done through sum and product operations, hence the name algebraic. Product types combine multiple types into one, such as pairs. Sum types allow a type to be one of more types. This is useful for representing types that can take more forms, such as the tree ADT in Figure 1. These product and sum types can be arbitrarily combined to create more complex types, and they can also be recursive.

ADTs can be polymorphic by parametrizing the type over a type variable. This type variable can be replaced with any specific type. For example, in Figure 1, we instantiate a tree of integers by replacing the type variable with `Int`. Polymorphism increases the flexibility and reusability of ADTs, as it removes the need to declare them for each type individually. This allows for more concise and maintainable code, as we can adapt the ADT to the context in which it is used.

Pattern matching can be used to deconstruct and manipulate ADTs. It allows us to access the different variants within an ADT and use its underlying values to perform calculations. Continuing with the tree example from before, we define a height function that is dependent on the form the ADT takes. Leaves are the end of the tree and, therefore, have a height of 1. For nodes, we can calculate the height by accessing their subtrees and recursively calling the height function.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)

-- example of a tree parameterized by Int
intTree :: Tree Int
intTree = Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)

-- example of pattern matching
height :: Tree a → Int
height (Leaf _) = 1
height (Node l r) = 1 + max (height l) (height r)
```

Figure 1: Algebraic data type in Haskell.

2.3 Agda

Agda is a purely functional, dependently typed, and total programming language. Dependent types allow us to use arbitrary values within types [8], which enables embedding properties the program must adhere to within the program itself. Dependent types also allow us to construct formal proofs in Agda [9] through the Curry-Howard correspondence, which means that aside from being a programming language, Agda serves as a proof assistant. Agda being a total language means programs are guaranteed to terminate.

One popular example [8; 10] showcasing the power of dependent types is that of `Vec A n` as seen in Figure 2. This is a polymorphic list which is dependent on its own length. `Vec` allows us to use the length of the list as a property on the type level. One specific use case is that of `head`, which returns the first element in the list. Without knowing the length of a list, we would have to wrap the result type of head in the `Maybe` data type since performing head on an empty list would fail. With `Vec`, we can reason that the list must be at least of length 1, which allows us to return the value directly.

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)

head : {A : Set} {n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

Figure 2: An implementation of `head` using `Vec` in Agda.

We can also use dependent types to encode relations between values at a type level. Figure 3 shows an inductively defined relation [3] for natural numbers. Its constructors form the base case and inductive case. We construct an instance of this relation by providing proof the relation holds. If it does not hold, we cannot construct this proof since the type has no inhabitants. Later, we will show that we can model the typing rules of our language as a relation.

```
data _<_ : ℕ → ℕ → Set where
  z<n : ∀ {n} → zero < suc n - base case
  s<s : ∀ {m n} → m < n → suc m < suc n - inductive case

proof : 1 < 2
proof = s<s z<n - constructing a proof
```

Figure 3: Inductive relation between natural numbers.

3 Toy Language

This section discusses the toy language used for the later implementation of the type-checker. We start by giving the syntax in Section 3.1. Then we discuss de Bruijn indices in Section 3.2. Finally, we discuss the type system in Section 3.3.

3.1 Language Syntax

The toy language used for this project includes ADTs, pattern matching, and polymorphism. The language itself is an extension of Girard's System F [11], which extends the STLC with explicit type abstractions and applications. We added case expressions and ADTs based on the Haskell syntax, as well as some minor language constructs such as naturals so that we can write an actual program. The syntax can be seen in Figure 4.

There are certain constraints on the declarations of new types, much like in Haskell. All type variables in $\overline{\alpha}$ in a type constructor $T\,\overline{\alpha}$ must be unique to avoid ambiguity. Furthermore, since all type variables in a data constructor's arguments are bound through the type constructor, we cannot use type variables as arguments that are not present in $\overline{\alpha}$. Finally, we reserve capitalized names for type and data constructors and require all types and data constructors to be unique.

Variables	x, y, z
Type variables	α, β
Type constructors	T
Data constructors	C
Programs	$\text{prog} ::= \overline{d}\ t$
Type Declaration	$d ::= \texttt{data}\ T\ \overline{\alpha}\ \texttt{where}\ \overline{C\ \overline{\sigma}}$
Atoms	$v ::= x \mid C$
Terms	$t, u, e ::= v \mid \lambda\, x : \sigma.t \mid \Lambda\alpha.t \mid t\, u \mid t\, \sigma$ $\mid \texttt{zero} \mid \texttt{suc}\ t \mid \texttt{true} \mid \texttt{false}$ $\mid \texttt{case}\ t\ \texttt{of}\ [\ \overline{p}\ ]$
Patterns	$p ::= C\ \overline{x}\ \rightarrow\ t$
Types	$\sigma, \tau, \xi, \phi ::= \forall\alpha.\sigma \mid \sigma \rightarrow \tau \mid T\ \overline{\sigma} \mid \alpha \mid \texttt{Bool} \mid \mathbf{N}$
Context	$\Gamma, \Delta ::= \varnothing \mid \Gamma, v : \sigma \mid \Gamma, \sigma$
Substitutions	$[\alpha \rightarrowtail \phi]$

Figure 4: Syntax of the toy language.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \texttt{VAR} \qquad \frac{\Gamma \vdash e : T\ \overline{\xi} \quad \Gamma \vdash^{ps} \overline{p} : T\ \overline{\xi} \rightarrow \tau \qquad \forall C \in \texttt{constructors}(T), \exists\, p_C \in \overline{p}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ [\ \overline{p}\ ] : \tau}\ \texttt{CASE}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad \Gamma \vdash^{k} \sigma}{\Gamma \vdash \lambda x : \sigma.e : \sigma \rightarrow \tau}\ \texttt{ABS} \qquad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}\ \texttt{APP}$$

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \texttt{T-ABS} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau \quad \Gamma \vdash^{k} \sigma}{\Gamma \vdash e\,\sigma : \tau[\alpha \rightarrowtail \sigma]}\ \texttt{T-APP}$$

$$\frac{\Gamma \vdash^{p} p : T\overline{\xi} \rightarrow \tau \qquad \Gamma \vdash^{ps} \overline{p} : T\ \overline{\xi} \rightarrow \tau}{\Gamma \vdash^{ps} p, \overline{p} : T\overline{\xi} \rightarrow \tau}\ \texttt{PCONS} \qquad \frac{}{\Gamma \vdash^{ps} \varnothing : T\overline{\xi} \rightarrow \tau}\ \texttt{PEMPTY}$$

$$\frac{(C : \forall\overline{\alpha'}.\overline{\sigma}^{c} \rightarrow\ T\ \overline{\xi}^{t}) \in \Gamma \quad \theta = \{\overline{\alpha \rightarrowtail \phi}\} \qquad \theta(\overline{\xi}) = \overline{\xi_p} \quad \Gamma, \overline{x : \theta(\sigma)}^{c} \vdash e : \tau}{\Gamma \vdash^{p} C\ \overline{x}^{c} \rightarrow e : T\ \overline{\xi_p} \rightarrow \tau}\ \texttt{PAT}$$

Figure 7: Typing rules of the toy language.

3.2 De Bruijn Indices

We use de Bruijn indices [12] to represent variable bindings in our language. De Bruijn indices are a common way of representing bindings namelessly. They remove the need to consider variable freshness constantly, as indices point directly to their corresponding binder. They also enforce α-equivalence, e.g., $\lambda x.x$ is equivalent to $\lambda y.y$ even though the variables' names differ. More importantly, they allow us to implement capture-avoiding substitution, which in our language is relevant for type variables in type applications.

Original expression
$e_1\ :\ \forall\alpha.\forall\beta.\alpha \rightarrow \alpha$

Type application *(applied β is free before being substituted)*
$e_1\ \beta\ :\ (\forall\beta.\alpha \rightarrow \alpha)[\alpha \rightarrowtail \beta]$

Resulting expression *(β is bound by the remaining type abstraction)*
$e_2\ :\ \forall\beta.\beta \rightarrow \beta$

Figure 5: Variable capture through type application.

Variable capture occurs when a variable becomes bound by a different binder upon substitution due to name clashes. An example can be seen in Figure 5 using notation from Figure 4. When a variable is captured, applications to its original binder will no longer replace this variable, as it is now bound elsewhere. This can cause subtle bugs in programs, and while they may not arise often, they pose a problem in the system. De Bruijn indices address this problem by eliminating name-wise comparisons between type variables, allowing for a substitution algorithm that does not require variable names to be unique.

One disadvantage of de Bruijn indices is that they can be hard to interpret. Since names are removed and only an index remains, finding where a variable is bound consists of counting back individual binders. This makes a complex program unreadable for humans. When we discuss term-level bindings in Section 4.1 we introduce a variation of de Bruijn indices which solves this problem.

3.3 Type System

The typing rules relevant to this paper can be seen in Figure 7. Most of the rules are taken directly from System F and are therefore not explained here. We exclude rules for type declarations as these only prepopulate the context with the constructors. Data constructors are function types that map their arguments to the newly declared type. If the type is polymorphic, we include type abstractions over all its type variables for every data constructor as seen in Figure 6. When there are multiple declarations the contexts are appended.

```
Declaration
data List a where Nil | Cons a (List a)

Context
∅, List a, Nil : ∀a.List a, Cons : ∀a.a → List a → List a
```

Figure 6: Declaration and conversion to a context.

Programmers may introduce types when annotating lambda arguments or applying a type to a type abstraction. We require these types to be well-kinded, which is enforced using the auxiliary judgment $\Gamma \vdash^{k} \sigma$ taken from Jones et al. [13] Kinds are the types of type constructors, and just like terms need to be well-typed, types need to be well-kinded. Concretely, this means that type variables need to be in scope. Also, ADTs must be declared and saturated, and their parameters must also be well-kinded. Saturation means the number of parameters matches the type constructor's arity.

The rules for pattern matching use the auxiliary judgments $\Gamma \vdash^{ps} \overline{p} : \sigma \rightarrow \tau$ and $\Gamma \vdash^{p} p : \sigma \rightarrow \tau$ based on Jones et al. [13], Chen and Erwig [14] and Jones et al. [15]. The PCONS rule works by iterating over a sequence of patterns, ensuring all patterns are valid and have the same return type. The PAT rule first asserts that C is a data constructor belonging to a type constructor T. A function θ replaces all free type variables in the constructor's arguments with their corresponding parameter from the scrutinee type of the case expression. Finally, we extend the context with the pattern variables bound to the transformed argument types and assert that the pattern body is well-typed in this context.

4 Implementation

Here, we present the implementation of the CbC type-checker for the toy language from the previous section. We first explain the required data types in Section 4.1. After that, we present the type-checker in Section 4.2.

4.1 Prerequisite Data Types

We first explain how we represent the syntax and how we split up the context. Finally, we combine these two and show how we modeled our typing rules as an inductive relation.

Representing Syntax

Our type-checker takes our program in the form of an abstract syntax tree. Agda's Unicode support allows us to write the different terms in a format that closely resembles our syntax. The most relevant syntax definitions can be seen in Figure 8.

```
data _∈_ (x : String) : Scope → Set where - ← membership relation
  here : {xs : Scope} → x ∈ (x :: xs)
  there : {y : String} {xs : Scope} → x ∈ xs → x ∈ (y :: xs)
data Type : Set where - ← types of terms
  TVar : ℕ → Type - ← variable using de Bruijn indices
  - ↓ ADT parameterized by a list of types
  T : String → List Type → Type
  - ...other types
mutual
  data Term (α : Scope) : Set where
    - ↓ variable using well-scoped names
    `_#_ : (x : String) → x ∈ α → Term α
    - ↓ lambda extends scope
    λ_:_⇒_ : (x : String) → Type → (v : Term (x :: α)) → Term α
    `case_of[_] : Term α → List (Pattern α) → Term α
    - ...other terms
  data Pattern (α : Scope) : Set where
    - ↓ pattern extends scope
    `_#_:_→_ : (x : String) → x ∈ α → (ns : List String)
               → Term (reverse ns ++ α) → Pattern α
```

Figure 8: A subset of the syntax definitions in Agda.

We use a variation of de Bruijn indices with names for term-level bindings. Since we have more ways of creating bindings for terms (constructors, lambda expressions, patterns), using regular de Bruijn indices can make the program hard to interpret. This is solved by adding names. We achieve well-scoped names [16] by providing the `Scope`[1] as a parameter and providing proof of membership when we access this scope. While this approach means we have to construct proofs manually, it provides the additional guarantee that there are no free variables by construction. An example program showcasing the difference between de Bruijn indices and well-scoped names can be seen in Figure 9.

```
term : Term Φ
term = Λ {-a-} Λ {-b-}
         (λ "f" : ({-a-} TVar 1 ⇒ {-b-} TVar 0) ⇒
           λ "x" : {-a-} TVar 1 ⇒
             ` "f" # there here · ` "x" # here)
```

Figure 9: Example with de Bruijn indices and well-scoped names.

[1] `Scope` is a type alias of `List String`.

Contexts

We split the context into two as seen in Figure 10. The `Context` holds all term-level bindings and associates them with their type. It is indexed on scope similarly to the terms discussed previously. The `TyContext` stores all declared types and is indexed on the number of type variables currently in scope. This second context is used to check well-kindedness, as defined in Section 3.3.

```
data Context : Scope → Set where
  ∅ : Context Φ
  - ↓ adding variable extends scope
  _,_:_ : {α : Scope} → Context α
        → (x : String) → Type → Context (x :: α)

data TyContext : ℕ → Set where
  ∅ : TyContext 0
  - ↓ adding new type to list of valid types
  _,_ : {n : ℕ} → TyContext n → Type → TyContext n
  - ↓ extending context with a type variable
  _,• : {n : ℕ} → TyContext n → TyContext (suc n)
```

Figure 10: Declaration of `Context` and `TyContext`.

Typing Rules

An essential component for our type-checker is the typing relation as defined in Figure 11. This data type is parametrized by the two contexts that were previously discussed and indexed over terms and types. It forms an inductive relation that states that a term evaluates to a valid type under a given context. We assume this context has been pre-populated with all declarations as described in Section 3.3, i.e., `Context` contains all data constructors, and `TyContext` contains all declared types. For the sake of conciseness, we omit the details of the implementation for converting declarations to a context and focus on type-checking terms.

```
data _;_⊢_:_ {n : ℕ} {α : Scope} (Γ : Context α) (Δ : TyContext n)
     : Term α → Type → Set where
  - ...other rules
  ⊢` - ↓ variable rule
    : {x : String}
    → (p : x ∈ α)
    → Γ ; Δ ⊢ ` x # p : lookupVar Γ x p
  ⊢λ - ↓ lambda rule
    : {x : String} {t₁ t₂ : Type} {e : Term (x :: α)}
    → Γ , x : t₁ ; Δ ⊢ e : t₂
    → Δ ⊢ᵏ t₁
    → Γ ; Δ ⊢ (λ x : t₁ ⇒ e) : t₁ ⇒ t₂
  ⊢∘ - ↓ type application rule
    : {t₁ t₂ : Type} {e : Term α}
    → Γ ; Δ ⊢ e : ∀' t₁
    → Δ ⊢ᵏ t₂
    → Γ ; Δ ⊢ e ∘ t₂ : t₁ [ t₂ ]
  ⊢case - ↓ case expression rule
    : {ts : List Type} {ps : List (Pattern α)} {t : Type}
      {e : Term α} {x : String}
    → Γ ; Δ ⊢ e : T x ts
    → Γ ; Δ ⊢ᵖˢ ps : T x ts → t
    - ↓ make sure there is a pattern for all constructors
    → (contextToConstructors Γ x) ↭ (patternsToConstructors ps)
    → Γ ; Δ ⊢ `case e of[ ps ] : t
```

Figure 11: A subset of the typing rules implemented in Agda.

The constructors in Figure 11 are the typing rules as outlined in Figure 7. They take derivations of strict subexpressions to construct a new derivation. They can also take additional properties, such as the auxiliary judgments that were introduced in Section 3.3. These auxiliary judgments are similarly translated into Agda from their definition in Figure 7 and are therefore not discussed here.

ADTs form one additional challenge for checking well-kindedness. The `TyContext` is a list of the declared types, and to check whether a data type is in scope and saturated, we must traverse the `TyContext` until we find the respective type. However, by traversing it, we discard types that are earlier in the list than the type we are looking for. If we then check that the parameters are also well-kinded, they could incorrectly be rejected if they reference a type that was earlier in the list. That is why we define an auxiliary data type that copies the original `TyContext`, as seen in Figure 12. The left copy is preserved for checking parameters, while the right copy is traversed in a style similar to that of the membership relation in Figure 8.

```
mutual
  data _⊢ᵏ_ {n₁ : ℕ} (Δ : TyContext n₁) : Type → Set where
    - ...other rules
    ⊢ᵏT - ↓ copies original context into one for traversing
      : {x : String} {ts : List Type}
      → Δ ; Δ ⊢ᵏ T x ts
      → Δ ⊢ᵏ T x ts
  data _;_⊢ᵏ_ {n₁ n₂ : ℕ} (Δ : TyContext n₁)
      : TyContext n₂ → Type → Set where
    ⊢ᵏT'
      : {Δ' : TyContext n₂} {x₁ x₂ : String} {ts₁ ts₂ : List Type}
      → x₁ ≡ x₂
      - ↓ verifies type is saturated
      → length ts₁ ≡ length ts₂
      - ↓ checks parameters (applies ⊢ᵏ to each parameter)
      → Δ ⊢ᵏˢ ts₂
      → Δ ; Δ' , T x₁ ts₁ ⊢ᵏ T x₂ ts₂
    ⊢ᵏthere - similar definition for ,·
      : {Δ' : TyContext n₂} {t₁ t₂ : Type}
      → Δ ; Δ' ⊢ᵏ t₁
      → Δ ; (Δ' , t₂) ⊢ᵏ t₁
```

Figure 12: Auxiliary data type for kind-checking ADTs.

4.2 Type-Checking Algorithm

Our type-checker constructs an instance of the typing relation introduced in Section 4.1, proving the relation holds. However, we cannot create an instance if the relation does not hold. Since Agda does not allow raising exceptions, we must handle ill-typed terms differently. We can use a monad to simulate the effect of raising an exception [17].

We use the `Evaluator` monad to handle programs for which our type-checker cannot construct a proof. It is an evaluator in the sense that it can evaluate whether the relation that is passed to it as a parameter holds. The `Evaluator` has two smart constructors, `evalError` which takes a string, and `eval0k` which takes the parameter type. This monad allows for continuous computation as long as no error is encountered. If an error is thrown, it stops computation and propagates the error message to the end. The `return` and `bind` operations for this monad are defined in Figure 13.

```
return : a → Evaluator a
return x = evalOk x

bind : Evaluator a → (a → Evaluator b) → Evaluator b
bind (evalError err) _ = evalError err
bind (evalOk x) f = f x
```

Figure 13: Pseudo-code of the return and bind functions for the `Evaluator` monad.

We use type inference to simultaneously reconstruct the type from a given term and verify that the constraints of the type system are adhered to. As mentioned in Section 2.1, type inference allows for fewer explicit type annotations. For example, instead of annotating the return type of a lambda expression and checking that the lambda body indeed returns that type, we can infer the body's type directly. Sometimes, we do need to check a term against a given type, for example, in function applications. We first infer the function type and then check that the supplied argument matches the function's expected argument type. This is done using `checkTerm`. The signatures for `inferTerm` and `checkTerm` can be seen in Figure 14. While both return a derivation for the given term, `inferTerm` returns it paired with the inferred type.

```
inferTerm : ∀ {α : Scope} {n : ℕ} (Γ : Context α) (Δ : TyContext n)
  (u : Term α) → Evaluator (Σ[ t ∈ Type ] Γ ; Δ ⊢ u : t)
checkTerm : ∀ {α : Scope} {n : ℕ} (Γ : Context α) (Δ : TyContext n)
  (u : Term α) (ty : Type) → Evaluator (Γ ; Δ ⊢ u : ty)
```

Figure 14: Sigatures of `inferTerm` and `checkTerm`.

The majority of the implementation can be found in `inferTerm`. For each term, we add a case that constructs an instance of its accompanying typing rule. The cases of `inferTerm` for the constructors from Figure 11 can be seen in Figure 15. Each case first constructs proofs of all relations that are required for its respective constructor. It does this by (recursively) calling evaluators that construct these relations. Then, it instantiates the typing relation by supplying its respective constructor with these proofs.

```
- ...other cases
inferTerm ctx tyCtx (` x # index ) = do
  return (lookupVar ctx x index , ⊢` index )
inferTerm ctx tyCtx (λ x : aTy ⇒ body) = do
  bTy , btd ← inferTerm (ctx , x : aTy) tyCtx body
  kind ← checkKind tyCtx aTy
  return (aTy ⇒ bTy , ⊢λ btd kind)
inferTerm ctx tyCtx (body ∘ t) = do
  (`∀ ty) , ltd ← inferTerm ctx tyCtx body
    where _ → evalError "invalid application"
  kind ← checkKind tyCtx t
  return (ty [ t ] , ⊢∘ ltd kind)
inferTerm ctx tyCtx (`case sc of[ ps ]) = do
  scTy@(T adt _) , sctd ← inferTerm ctx tyCtx sc
    where _ → evalError "can not pattern match on non-adt"
  t , pstd ← inferPatterns ctx tyCtx ps scTy
  eq ← evalSetEquiv
    (contextToConstructors ctx adt) (patternsToConstructors ps)
  return (t , ⊢case sctd pstd eq)
```

Figure 15: Different cases of the type inference algorithm.

Due to intrinsic typing, each valid implementation for the evaluators is sound with respect to the typing relation. As mentioned in Section 2.3, we cannot construct a proof for a relation if the relation does not hold. Therefore, an evaluator constructing an instance of the relation can also never create one for a term that does not adhere to the typing rules. This also means that if we were to make a mistake that would make the type-checker unsound, the type-checker would not compile. For example, if we compare the implementation of `inferTerm` for a lambda expression to the definition of the lambda rule in Figure 11, we see that the recursive call correctly extends the context with the argument before calling `inferTerm` on the lambda body. If we had not done this, the resulting proof would have been for a different relation than required by the constructor, preventing the type-checker from compiling. This is what we refer to as correct-by-construction. The type definitions effectively guide us in providing a valid implementation.

```
checkKind : ∀ {n : ℕ} (Δ : TyContext n) (t : Type) → Evaluator (Δ ⊢ᵏ t)
checkKind' : ∀ {n n' : ℕ} (Δ : TyContext n) (Δ' : TyContext n') (t : Type)
  → Evaluator (Δ ; Δ' ⊢ᵏ t)
inferPattern : ∀ {α : Scope} {n : ℕ} (Γ : Context α) (Δ : TyContext n)
  (p : Pattern α) (s : Type) → Evaluator (Σ[ t ∈ Type ] Γ ; Δ ⊢ᵖ p : s → t)
inferPatterns : ∀ {α : Scope} {n : ℕ}
  (Γ : Context α) (Δ : TyContext n) (ps : List (Pattern α)) (s : Type)
  → Evaluator (Σ[ t ∈ Type ] Γ ; Δ ⊢ᵖˢ ps : s → t)
```

Figure 16: Function signatures for the other evaluators.

Similarly to how we defined an evaluator for `inferType`, we define evaluators for the auxiliary judgments and relations we require, as seen in Figure 16. The `checkKind` evaluator takes the current `TyContext` and a type. The `inferPatterns` and `inferPattern` take the current context, the pattern(s) to check, and the inferred scrutinee type of the case expression, as seen in Figure 15. The implementations of these evaluators are similar to that shown in Figure 15, and therefore, they are not discussed in detail here.

5 Discussion

This section discusses the findings of this paper. We first discuss the type-checker in Section 5.1 and refer back to soundness and completeness as defined in the introduction. After that, we reflect on the challenges of the CbC approach in Section 5.2. Finally, we discuss challenges specific to Agda in Section 5.3.

5.1 Soundness and Completeness

In the previous section, we demonstrated how to translate the formalization of the language in Section 3 into Agda and construct a type-checker for it. An example of how the type-checker is called can be seen in Figure 17. We demonstrated that due to intrinsic typing, the implementation of the algorithm is sound with respect to the typing relation, with the proof being embedded in the program itself. Additionally, due to the nature of Agda, the algorithm is guaranteed to terminate, and no run-time exceptions can occur when executing the type-checker. Assuming that the typing rules were correctly translated into Agda and that auxiliary functions, such

```
ctx : Context ("Cons" :: "Nil" :: [])
ctx = (∅
  , "Nil" : ‘∀ (T "List" (TVar 0 :: []))
  , "Cons" : ‘∀ (TVar 0 ⇒ T "List" (TVar 0 :: []) ⇒ T "List" (TVar 0 :: []))

tyCtx : TyContext 0
tyCtx = ∅ , T "List" (TVar 0 :: [])
- indices are replaced with {-} for the sake of conciseness
term : Term ("Cons" :: "Nil" :: [])
term =
  (Λ
    (λ "x" : TVar 0 ⇒
    ‘ "Cons" # {-} ∘ TVar 0 · ‘ "x" # {-} · (‘ "Nil" # {-} ∘ TVar 0)
    )) ∘ ‘ℕ · ‘zero

singleton : Evaluator (ctx ; tyCtx ⊢ term : T "List" (‘ℕ :: []))
singleton = checkTerm ctx tyCtx term (T "List" (‘ℕ :: []))
```

Figure 17: Example of type-checking a well-typed term.

as the substitution algorithm, are correct, our type-checker cannot accept ill-typed terms. While this assumption could be proven, it was deemed outside this project's scope. Nonetheless, the type-checker successfully handled all 19 test cases, which included 6 well-typed terms and 13 ill-typed terms.

Some might argue that a similar guarantee of soundness can be achieved through testing. However, the guarantee provided by the CbC approach is fundamentally different. As Dijkstra famously said, "Program testing can be used to show the presence of bugs, but never to show their absence" [18]. One study showed that typing bugs are prevalent in JVM compilers, with some being attributed to soundness issues [19]. While extensive testing might not uncover all these bugs, the previous section demonstrated that intrinsic typing prevents us from constructing an unsound implementation altogether. Therefore, when soundness is required, the CbC approach offers guarantees traditional testing cannot.

One limitation of the type-checker presented in this paper is that correct programs might incorrectly get rejected. i.e., the type-checker may be incomplete. As Wadler et al. mentioned, "nothing prevents us from writing a function that always returns an error, even when there exists a correct derivation" [3]. While this mostly requires deliberate decisions, it could also happen by mistake. A stronger guarantee of correctness could be achieved by also providing a proof in the negative case, i.e., proof that a program cannot be well-typed. This can be done by returning a decidable instance of the relation instead of an evaluator, as shown in Figure 18. Due to time constraints, it was decided that adding completeness to the type-checker was outside this project's scope.

```
- ↓ incomplete but sound implementation for eval<
eval< : (m n : ℕ) → Evaluator (m < n)
eval< m n = evalError "Something went wrong"

- ↓ complete and sound implementation for decidable _<_
dec< : (m n : ℕ) → Dec (m < n)
dec< zero (suc n) = yes z<n
dec< (suc m) (suc n) with dec< m n
... | yes p = yes (s<s p)
... | no ¬p = no λ {(s<s p) → ¬p p}
dec< (suc m) zero = no λ ()
dec< zero zero = no λ ()
```

Figure 18: Incomplete implementation of an evaluator and implementation for decidable for the relation defined in Section 2.3.

5.2 Challenges of CbC Programming

The CbC approach's main challenge is the added complexity due to using dependent types. When we index a type, this may introduce dependencies that the programmer needs to maintain, which makes extending the system and refactoring more complex. For example, with well-scoped names, we need to ensure the context and terms are compatible and each extends the scope appropriately. This requires additional complexity at a type level in the Agda program when declaring these data types and using them alongside each other. In this case, we argue that the added complexity is worth it. It ensures no free variables by construction, and that context and terms are appropriately combined in the judgment data type. However, it is a consideration to keep in mind.

Additionally, encoding the full behavior for all functions within their signature can be challenging, meaning some parts of the program may still require a separate proof of correctness. An example of this in our project is the substitution algorithm for type variables, which is not intrinsically verified. While the algorithm is correct as it is based on existing solutions, a mistake could still be made while translating it to Agda. To still provide a stronger guarantee of correctness than traditional testing can offer, we can use Agda to prove that the function is correct. However, this would then not be encoded within the function itself. As mentioned in Section 5.1, proving the correctness of all auxiliary functions was considered outside this project's scope.

5.3 Challenges of Agda

A consideration specific to Agda is that the termination checker can feel restrictive at times. Agda guarantees termination by requiring recursive calls to operate on strict subexpressions of the arguments [9]. However, certain programs fail the termination check when equivalent programs are accepted. One such example can be seen in Figure 19. The first program does not pass the termination check, while the second one does. The solution here was to implement a specialized map function, which violates the DRY principle [20], stating that we should not repeat ourselves in code. Problems with termination checking can also arise when using `with` abstractions [9]. These restrictions can be hard to work around and frustrating if the programmer is certain programs terminate.

```
-- ↓ does not pass the termination check
show : Type → String
show (T n ts) = n ++ "  " ++ (concat $ map show ts)

-- ↓ does pass the termination check
show : Type → String
show' : List Type → List String

show (T n ts) = n ++ "  " ++ (concat $ show' ts)

show' [] = []
show' (t :: ts) = show t :: show' ts
```

Figure 19: Two equivalent programs, lower terminates, upper does not.

We used the Agda standard library to speed up development, which contains definitions for commonly used constructs like lists, monads, and more. One disadvantage of the standard library is that it can be hard to navigate. The documentation [21] of the standard library lacks search functionality, and with hundreds of modules, it can be challenging for a developer to find the desired item. Furthermore, many definitions reference types and functions from other modules, which forces programmers to go through multiple modules before understanding how to use a certain item. Another consideration when using the standard library is that it was not created with computation in mind but rather for ease of proof. If computational performance is important, developers may have to look for a different solution.

6 Responsible Research

All code produced for this research is available in a public repository[2]. While we believe this paper discussed all relevant concepts to understand the findings of this research, many details of the implementation were omitted for the sake of conciseness. By providing the full source code, we ensure transparency of the work and allow for reproducibility. Alongside the source code, the reader can find documentation on how to install and run the program, as well as more documentation on the specifics of the implementation and some example programs in the toy language.

All content of this paper was produced by the author, with proper citations and acknowledgments for external contributions. While ChatGPT[3] assisted in rewriting some parts of the paper for clarity, this was done in accordance with the TU Delft's policy on LLMs. The typical query that was used was: "Rewrite this paragraph/sentence such that it is easier to understand." Results produced by ChatGPT were never directly put in the paper. Instead, they served as inspiration for improving certain paragraphs, and the rewriting was always done manually in the author's words.

7 Conclusions and Future Work

This section gives the conclusions of this paper in Section 7.1. After that, we suggest some areas for future work in Section 7.2.

7.1 Conclusions

This paper presents a CbC type-checker written in Agda. It was written for a toy language with ADTs, pattern matching, and polymorphism. The type-checker constructs a proof of well-typedness for any given term using typing rules taken directly from the language's formalization. Since it cannot construct a proof for ill-typed terms, the type-checker is considered to be sound with respect to the typing rules.

We have shown that using the CbC approach offers significant benefits. Due to intrinsic typing, the proof of soundness is embedded in the implementation of the type-checker itself. This also means that the types effectively guide the implementation since an implementation that does not conform to the typing rules will not compile. Other benefits of the CbC approach in Agda are that no run-time errors can occur when

[2]https://github.com/MilosRistic02/cbc-adt-type-checker
[3]https://chatgpt.com/

executing the type-checker, and the execution is guaranteed to terminate.

Nonetheless, some challenges accompany the CbC approach in Agda. Encoding properties at a type level introduces added complexity, making the code harder to maintain. Furthermore, finding ways to encode behavior within a function's signature can be challenging, meaning some parts of the program may still require a separate proof of correctness. A consideration specific to Agda is that the termination checker can be restrictive since it requires recursive calls to be made on strict subexpressions. This means that some programs that are known to terminate are incorrectly rejected, and working around this can be challenging. Finally, the current documentation for the standard library of Agda can be hard to navigate.

7.2 Future Work

This paper is centered around the toy language presented in Section 3, which includes interesting features such as polymorphism, ADTs, and pattern matching. However, it lacks support for many features that are required to make the language more practical. Examples of such features include recursion, let bindings, modules, and many more. It is worth exploring how the type-checking algorithm could be extended to accommodate more complex languages by incorporating these features incrementally.

One closely related feature to those discussed in this paper is that of generalized algebraic data types (GADTs). GADTs are available as a language extension in Haskell and offer interesting functionality. Although their syntax differs only slightly from the ADTs presented in this paper, their type inference becomes significantly more complex. The typing rules in Section 3 were based on research on type inference for GADTs [13; 14; 15], and so extending this work for GADTs seems like a logical next step.

The type-checker itself is only part of a broader system that includes the parser and compiler. Implementing a complete system using the CbC approach could be beneficial, as it offers similar guarantees to the ones presented in this paper. Rouvoet [10] explored CbC implementations for compilers and interpreters, but there remains a gap in merging these implementations with the style of type-checking presented in this paper. It would be worth exploring whether a complete implementation could be achieved and whether the benefits of such a system justify the additional challenges it may bring.

8 Acknowledgements

I want to thank my supervisors, Jesper Cockx and Sára Juhošová, for their guidance and feedback throughout the project. I am particularly thankful for the example type-checker that was provided at the start of this project. It helped shape my initial understanding of this area of research and served as the foundation for my implementation.

Additionally, I want to thank the other members of my research group for discussing ideas and sharing their findings, which significantly contributed to the progress of my work. Finally, I would like to thank my parents for supporting me.

References

[1] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[2] B. C. Pierce, *Advanced Topics in Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2004.

[3] P. Wadler, W. Kokke, and J. G. Siek, *Programming Language Foundations in Agda*. Aug. 2022. [Online] Available: https://plfa.inf.ed.ac.uk/22.08/.

[4] J. Cockx. (2019). Correct-by-construction programming in Agda: indexed datatypes and dependent pattern matching [Online]. Available: https://jespercockx.github.io/ohrid19-agda/slides/slides2.html#/title-slide.

[5] K. I. Casamento, "Correct-by-Construction Typechecking with Scope Graphs," M.S. thesis, Comput. Sci., Portland State Univ., Portland, OR, 2019.

[6] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau and T. Winterhalter, "Coq Coq correct! verification of type checking and erasure for Coq, in Coq," in *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Article 8, Jan. 2020, pp 1-28, doi: 10.1145/3371076.

[7] L. Cardelli, "Type Systems," in *CRC Handbook of Computer Science and Engineering*, A. B. Tucker, 2nd ed., Boca Raton, FL, USA: CRC Press, 2004, ch. 97.

[8] U. Norell, "Dependently Typed Programming in Agda," in *Advanced Functional Programming*, P. Koopman, R. Plasmeijer, D. Swierstra, Berlin, Germany: Springer, 2009, pp. 230–266.

[9] The Agda Team. "Agda 2.4.6.3 Documentation." agda.readthedocs.io. Accessed: May 23, 2024. [Online] Available: https://agda.readthedocs.io/en/v2.6.4.3-r1/.

[10] A. Rouvoet, "Correct by Construction Language Implementations," Ph.D. dissertation, TU Delft, Delft, NL, 2021.

[11] J. Girard, "The system F of variable types, fifteen years later," *Theor. Comput. Sci.*, vol. 45, no. 2, pp. 159–192, Sep. 1986.

[12] N. G. de Bruijn, "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem," *Indagationes Mathematicae*, vol. 75, no 5. pp. 381-392, 1972.

[13] S. P. Jones, G. Washburn, and S. Weirich, "Wobbly types: type inference for generalised algebraic data types," Univ. Pennsylvania, Philadelphia, PA, USA, Tech. MS-CIS-05-26, Jul. 2004.

[14] S. Cheng and M. Erwig, "Principal Type Inference for GADTs," *SIGPLAN Not.*, vol. 51, no. 1, pp. 416-428, Jan. 2016, doi: 10.1145/2914770.2837665.

[15] S. P. Jones, D. Vytionitis and G. Washburn, "Simple unification-based type inference for GADTs," *SIGPLAN Not.*, vol. 41, no. 9, pp. 50-61, Sep. 2006, doi: 10.1145/1160074.1159811.

[16] J. Cockx. “1001 Representations of Syntax with Binding.” jesper.sikanda.be. Accessed: May 28, 2024. [Online] Available: https://jesper.sikanda.be/posts/1001-syntax-representations.html.

[17] P. Wadler, “The essence of functional programming,” *Proc. 19th SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1992, pp. 1–14, doi: 10.1145/143165.143169.

[18] O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*. London, UK: Academic, 1972.

[19] S. Chaliasos, T. Sotiropoulos, G.-P. Drosos, C. Mitropoulos, D. Mitropoulos and D. Spinellis, “Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers,” in *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Article 123, Oct. 2021, pp 1-30, doi: 10.1145/3485500.

[20] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Harlow, UK: Addison-Wesley, 1999.

[21] Agda standard library. “Documentation for the Agda standard library.” agda.github.io. Accessed: Jun. 5, 2024. [Online] Available: https://agda.github.io/agda-stdlib/.