Correct-by-construction Type Checking for Substructural Type Systems

Vince Szabó[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 22, 2024

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

Abstract. As programming languages become ever more sophisticated, there is growing interest in powerful and expressive type systems. Substructural type systems increase expressiveness by allowing the programmer to reason about the number of times and the order in which variables can be used. This can be used to model different kinds of memory allocation and to construct more expressive interfaces. However, implementing complex type systems requires complex type checkers – this complexity increases the chances of bugs in the type checker itself, which could lead to invalid programs being accepted or valid programs being rejected. The consequences of such bugs can range from programmer frustration to critical errors in production systems.

In this thesis, we develop a correct-by-construction type checker for a toy language with a substructural type system. We use Agda's dependent type system to intrinsically ensure the soundness and completeness of the type checker. We discuss the advantages and disadvantages of the correct-by-construction approach and find that its complexity likely restricts it to specific use cases.

1 INTRODUCTION

Many programming languages include static checks to guarantee certain properties of the program without having to run tests. Many of these static checks are enforced by the type system. As languages evolve, these guarantees are becoming stronger and more granular, leading to more complex type systems. This complexity increases the probability of bugs in type checkers. Such bugs, while rarely triggered, can be devastating as they undermine confidence in the language, and hence in the software written in the language. Preventing such bugs is particularly important in proof assistants, where correct programs correspond to true logical statements, and safety-critical areas, where bugs can cause death or injury.

Many software quality assurance methods can be applied to type checkers – we specifically investigate *correct-by-construction programming*, a form of intrinsic verification where the type system of the implementation language is used to ensure correctness. Unlike most other quality assurance methods, correct-by-construction programming promises full confidence in the correctness of the code, but in return, it also requires that the code be written in a specific way. In this work, we examine the implications of this tradeoff.

Correct-by-construction programming. For the purposes of this paper, the term *correct-by-construction (CbC) programming* is defined as a method of programming in which types are used to create a strict specification for program behaviour, which is checked by the type checker. This requires an implementation language with an expressive type system. We use Agda [1], a dependently typed functional language designed for use as a proof assistant.

To illustrate the core ideas of correct-by-construction programming, we consider the example of finding an element in a list that satisfies a given predicate. Figure 1 shows two possible types for this function. The first one (`find`) is how such a function would usually be written in a functional language: it takes a parameter specifying the predicate and list and returns a `Maybe` containing either the appropriate value or `nothing`. Characteristically for Agda, the function also takes the type (denoted `Set`) `A` of the list values as an implicit parameter (denoted by curly braces). While the type signature is correct, it is trivial to write an incorrect implementation of `find`, for example, by always returning the first element of the list regardless of whether the predicate holds for it or not.

```
find : {A : Set} → (A → Bool) → List A → Maybe A

find' : {A : Set} {P : Pred A} → Decidable P → List A → Maybe (Σ[ x ∈ A ] P x)
```

Fig. 1. Two type signatures for a `find` function in Agda: one not in CbC style and one in CbC style

The second type signature, `find'` is written with the correct-by-construction programming method. Instead of using a Boolean function as the predicate type, it takes two separate arguments:

Type	Abbrev.	Structural rules	Usage restrictions
Unrestricted	`un`	Weakening, contraction, exchange	–
Affine	`aff`	Weakening, exchange	At most once
Relevant	`rel`	Contraction, exchange	At least once
Linear	`lin`	Exchange	Exactly once
Ordered	`ord`	–	Exactly once, respecting order

Table 1. An overview of types based on the allowed structural rules

a type `P` denoting a proof for the predicate (`Pred A`), and a function that *decides* whether the predicate holds or not, providing a proof along with its decision. The return value can be read as *"there exists some* `x` *of type* `A` *such that* `P x` *holds"*. It is an example of a dependent pair, with the first element being the value satisfying the predicate, and the second element being a proof of this satisfaction. Thanks to this proof, this type makes it impossible to write an implementation of `find'` that returns an incorrect element.

Substructural type systems. Substructural type systems allow programmers to express constraints on the number of times and the order in which variables can be used. The name arises from the fact that the ability in usual type systems to use variables any number of times in any order is due to three so-called structural rules: weakening, contraction, and exchange [15]. Removing one or more of these structural rules leads to a substructural type system. There are four main kinds of substructural types: *affine* types, which can be used at most once, *relevant* types, which must be used at least once, *linear* types, which must be used exactly once, and *ordered* types, which must be used exactly once, in reverse order of declaration *w.r.t.* other ordered variables. An overview of substructural types is presented in Table 1. Substructural type systems are useful for formalizing memory management [5, 7, 10] and concurrency [13], and can be used to design more expressive software interfaces [14]. A prominent example of a general-purpose programming language using substructural types is Rust, whose types are affine by default.

Contributions and structure. The main contributions of this thesis are to provide a description and a software artifact for a correct-by-construction type checker for substructural type systems, and to present the advantages, disadvantages, and possible applications of such type checkers. Section 2 introduces the example language and the theoretical background for the type checker. Section 3 describes the Agda implementation of the type checker. Section 4 discusses the merits of correct-by-construction type checkers. Section 5 addresses responsible research, and Section 6 gives a brief overview of related work. Finally, Section 7 closes the paper.

2 METHOD

To investigate the potential use cases of correct-by-construction type checkers for substructural type systems, we develop one in Agda for a small language with a substructural type system called λ_{sub}. Section 2.1 describes the programming language, Section 2.2 details its type system, and Section 2.3 gives a high-level description of the type checker.

2.1 The example language

λ_{sub}, the language targeted by our type checker, is based on the ordered lambda calculus described by David Walker [15]. We extend Walker's calculus with affine and relevant types to cover all substructural types, as well as a unit type, which makes some examples clearer. The language features were chosen such that the language is sufficiently complex to showcase the properties of substructural types, but also simple enough to be quickly understandable for a reader familiar with the simply typed lambda calculus.

Figure 2 presents a few examples of programs in λ_{sub}. All types and literals are annotated with a qualifier (`un`/`aff`/`rel`/`lin`/`ord`), which specifies whether the type should be substructural and if

$$\text{un } \lambda cond : (\text{un Bool}).\, \text{let } x := \text{aff unit } \Rightarrow \text{ if } cond \text{ then (eat } x) \text{ else (un unit)}$$

(a) A valid program using affine variables

$$\text{un } \lambda cond : (\text{un Bool}).\, \text{let } x := \text{lin unit } \Rightarrow \text{ if } cond \text{ then (eat } x) \text{ else (eat } x)$$

(b) A valid program using linear variables

$$\text{un } \lambda x : \text{ord (ord Unit} \times \text{ord Bool}).\, \text{split } x \text{ as } a, b \Rightarrow \text{ if } b \text{ then (eat } a) \text{ else (eat } a)$$

(c) A valid program using ordered variables

$$\text{un } \lambda x : \text{ord (ord Bool} \times \text{ord Unit}).\, \text{split } x \text{ as } a, b \Rightarrow \text{ if } a \text{ then (eat } b) \text{ else (eat } b)$$

(d) An ill-typed program using ordered variables

Fig. 2. λ_{sub} examples

so, what kind. The unit type represents some arbitrary resource and can be constructed using the `unit` literal.

For substructural types to be well-defined, it must be clear what constitutes a "use" of a value. In λ_{sub}, Booleans can be used as conditions in `if` terms, units are used by `eat` terms, pairs are used by `split` terms, and functions are used by applying them. Furthermore, passing a value of any type to a function, returning it from a function, or returning it at the top level constitutes a use.

Variables that must be used (*i.e.* variables with relevant, linear, or ordered type) must be used in all control paths, as seen in Figures 2b and 2c. Dually, variables that can only be used once (most importantly affine variables) are considered consumed if they are used in any control path.

A number of limitations are carried over from Walker's calculus. Ordered functions are not allowed, and pair construction and function application require arguments to be variables instead of arbitrary terms (*e.g.* $f\, x$ is allowed, but $f\, (g\, x)$ is not). This is mainly to simplify interactions with ordered terms.

2.2 The type system of λ_{sub}

Walker's work [15] provides declarative typing rules for his ordered lambda calculus, as well as algorithmic typing rules for the linear part of the calculus. Declarative typing rules, while more amenable to analysis, are hard to use for type checking, as they often contain nondeterministic choice. Algorithmic typing rules are instead deterministic and syntax-directed, meaning they can be used to implement a type checker by pattern matching on a term.

We extend Walker's typing rules with algorithmic versions of the rules for the ordered calculus and make changes necessary to add affine and relevant types. Our work considers these algorithmic rules the ground truth for λ_{sub}'s type system. Figure 3 shows some example typing rules.

The typing relation is of the form $\Gamma \vdash t : T, \Gamma' ; \Omega$, read "in context Γ, term t has type T, with output context Γ' and usage context Ω". The input context Γ keeps track of the types of all available variables before evaluating the term, while the output context Γ' stores variables available after evaluation. The usage context Ω tracks relevant variables that have been used at least once in the term – this is necessary, as relevant variables may be reused, and thus cannot be removed from Γ, but must be used at least once, thus we must record that they have been used. Contexts are (ordered) lists of variable bindings.[1]

We present a few examples to demonstrate the principles of the typing rules. There is one typing rule for variables of each possible substructural type. TUVar handles unrestricted variables – given a binding with type un T for variable x, it assigns the term x the type un T, leaving the context unchanged. TRVar works similarly with relevant types, except it also records the variable as used. In the case of TOVar, the output context is different from the input context, in that the used variable's binding is removed. This is expressed using the binding deletion relation $\Gamma - (x \mapsto T) = \Gamma'$. The

[1]Usage contexts could be represented as lists of variables, as the types of variables in them are irrelevant. However, using lists of bindings for all contexts allows us to use the same context intersection relation $\cap_{\mathsf{q}}$ (described below) for all contexts.

$$\frac{(x \mapsto \mathsf{un}\ T) \in^* \Gamma}{\Gamma \vdash x : \mathsf{un}\ T, \Gamma ; \varnothing}\text{TUVar} \qquad \frac{(x \mapsto \mathsf{rel}\ T) \in^* \Gamma}{\Gamma \vdash x : \mathsf{rel}\ T, \Gamma ; \varnothing, (x \mapsto \mathsf{rel}\ T)}\text{TRVar}$$

$$\frac{(x \mapsto \mathsf{ord}\ T) \in^* \Gamma \qquad \Gamma - (x \mapsto \mathsf{ord}\ T) = \Gamma'}{\Gamma \vdash x : \mathsf{ord}\ T, \Gamma' ; \varnothing}\text{TOVar}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathsf{q}\ \mathsf{Bool}, \Gamma_2 ; \Omega_1 \\ \Gamma_2 \vdash t_2 : T, \Gamma_3 ; \Omega_2 \qquad \Gamma_2 \vdash t_3 : T, \Gamma_4 ; \Omega_3 \\ \Gamma_3 \cap_{\mathsf{aff}} \Gamma_4 = \Gamma_5 \qquad \Omega_2 \cap_{\mathsf{rel}} \Omega_3 = \Omega_4 \qquad \Omega_1 \cup \Omega_4 = \Omega_5\end{array}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T, \Gamma_5 ; \Omega_5}\text{TIf}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1, \Gamma_2 ; \Omega_1 \qquad \Gamma_2, x \mapsto T_1 \vdash t_2 : T_2, \Gamma_3 ; \Omega_2 \\ \Gamma_3 ; \Omega_2 \div \varnothing, x \mapsto T_1 = \Gamma_4 \qquad \Omega_1 \cup \Omega_2 = \Omega_3\end{array}}{\Gamma \vdash \mathsf{let}\ x \coloneqq t_1 \Rightarrow t_2 : T_2, \Gamma_4 ; \Omega_3}\text{TLet}$$

Fig. 3. Some typing rules for λ_{sub}

ordering of variables is enforced implicitly by the context inclusion relation $(x \mapsto T) \in^* \Gamma$ – if T is an ordered type, then it must be the rightmost ordered binding in Γ, otherwise the inclusion does not hold. The typing rules for affine and linear variables work analogously, with both of them deleting the variable binding from the context to prevent the reuse of the value.

In terms with multiple subterms, the different contexts must be combined correctly. In the case of if terms, this is handled by the TIf rule. Since the condition is always evaluated first, its output context is used to type check the branches. This way information about variables used up in the condition is passed to the branches. Each branch may produce a different output and usage context by using different variables. The output contexts from the two branches are unified using the $\cap_{\mathsf{aff}}$ relation. The proposition $\Gamma_1 \cap_{\mathsf{aff}} \Gamma_2 = \Gamma_3$ holds if and only if Γ_1, Γ_2, and Γ_3 agree on all variables except affine ones, which must be present in Γ_3 if and only if they are present in both Γ_1 and Γ_2. The effect of this definition is that linear and ordered variables must be used by either both or neither branches, while affine variables may also be used by only one, in which case they are considered used up by the if term. Note that relevant and unrestricted bindings are always preserved by this unification, as they are not deleted when their variable is used. As for the usage contexts, there are three that need to be unified: Ω_1, containing the relevant variables used by the condition, and Ω_2 and Ω_3, containing the relevant variables used by each branch. Relevant variables must be used in all control paths, thus we take the intersection ($\cap_{\mathsf{rel}}$) of Ω_2 and Ω_3, keeping only variables used by both branches and then take the union of the result with Ω_1, since any variables used by the condition are used by the if term as a whole.

The rules presented previously track the usages and prohibit the reuse of variables with the appropriate types, however, they do not enforce the usage of variables with appropriate types. This is done at the point at which the variables are bound – TLet is provided as an example. Most of the work is done by the context division relation $\Gamma_1 ; \Omega \div \Gamma_2 = \Gamma_3$, which relates a typing context Γ_1, a usage context Ω, and a context Γ_2 listing bindings created by the current term with an output context Γ_3.

The purpose of context division is to enforce the usage of relevant, linear, and ordered variables, and to uphold the scoping rules by deleting the appropriate bindings. The definition for context division is given in Figure 4, extending Walker's definition [15] with affine and relevant types. Unrestricted bindings are simply removed as seen in rule DUn. Affine bindings may or may not be used – if used, the context is unchanged (DAff1), if not used, the binding is deleted to uphold scoping rules (DAff2). Usage of relevant bindings is enforced by requiring them to be in the usage context Ω (DRel). Finally, linear and ordered bindings are handled by the DMustUseOnce rule, which enforces their usage by requiring their absence from the output context.

$$\frac{}{\Gamma\ ;\ \Omega \div \varnothing = \Gamma}\ \text{DEmpty}$$

$$\frac{\Gamma_1\ ;\ \Omega \div \Gamma_2 = \Gamma_3 \qquad (x \mapsto \text{aff}\ T) \notin^* \Gamma_3}{\Gamma_1\ ;\ \Omega \div \Gamma_2,\ (x \mapsto \text{aff}\ T) = \Gamma_3}\ \text{DAff1}$$

$$\frac{\Gamma_1\ ;\ \Omega \div \Gamma_2 = \Gamma_3 \qquad (x \mapsto \text{rel}\ T) \in^* \Gamma_3 \qquad (x \mapsto \text{rel}\ T) \in^* \Omega \qquad \Gamma_3 - (x \mapsto \text{rel}\ T) = \Gamma_4}{\Gamma_1\ ;\ \Omega \div \Gamma_2,\ (x \mapsto \text{rel}\ T) = \Gamma_4}\ \text{DRel}$$

$$\frac{\Gamma_1\ ;\ \Omega \div \Gamma_2 = \Gamma_3 \qquad (x \mapsto \text{un}\ T) \in^* \Gamma_3 \qquad \Gamma_3 - (x \mapsto \text{un}\ T) = \Gamma_4}{\Gamma_1\ ;\ \Omega \div \Gamma_2,\ (x \mapsto \text{un}\ T) = \Gamma_4}\ \text{DUn}$$

$$\frac{\Gamma_1\ ;\ \Omega \div \Gamma_2 = \Gamma_3 \qquad (x \mapsto \text{aff}\ T) \in^* \Gamma_3 \qquad \Gamma_3 - (x \mapsto \text{aff}\ T) = \Gamma_4}{\Gamma_1\ ;\ \Omega \div \Gamma_2,\ (x \mapsto \text{aff}\ T) = \Gamma_4}\ \text{DAff2}$$

$$\frac{\Gamma_1\ ;\ \Omega \div \Gamma_2 = \Gamma_3 \qquad (x \mapsto \text{q}\ T) \notin^* \Gamma_3 \qquad \text{q} = \text{lin} \vee \text{q} = \text{ord}}{\Gamma_1\ ;\ \Omega \div \Gamma_2,\ (x \mapsto \text{q}\ T) = \Gamma_3}\ \text{DMustUseOnce}$$

Fig. 4. Definition of context division

As described, the typing relation effectively produces two outputs: the output context Γ' and the usage context Ω. It is possible to combine these two into a single list of bindings that stores a usage status for each variable. Then, instead of deleting bindings when a variable is consumed, we could set their status to "used". Used variables would then be ignored by the inclusion relation used in the variable typing rules. We ultimately decided against this approach as it would require defining more relations than the current solution, and would likely make the typing rules more complicated.

Due to the coexistence of multiple kinds of substructural types, some care is required when reasoning about terms that can contain other terms: namely pairs and lambda abstractions (which contain terms in their closures). Like Walker [15], we solve this by defining a partial order on qualifiers, ordering them by restrictiveness: for example, un is less restrictive than aff, and aff is less restrictive than lin. When creating a term of a composite type, all components must have less or equally restrictive qualifiers than the composite itself (*e.g.* one cannot put a term of linear type inside an unrestricted pair). This is enforced via the $q\ \langle T \rangle$ and $q\ \langle\!\langle \Gamma \rangle\!\rangle$ relations, which hold if the given type T or context Γ can be contained within a composite type of the qualifier q.

2.3 An overview of the type checker

A type checker is, in essence, a function that, given a term, returns the type of the term or signals failure if the term is ill-typed.[2] In practice, the type checker also takes the typing context as input, and, in our case, it also returns an output typing context and usage context. These inputs and outputs are all domains of the typing relation, and type checking can be seen as the process of finding the outputs that relate to the input.

For us to consider it correct, the type checker must be *sound* and *complete*. Soundness means that if the type checker returns a type for a term, this type must be correct – in other words, there must be a derivation for this type using the typing rules of the language. Completeness means that if the type checker claims that the term is ill-typed, it must indeed be so – that is, there must be no typing derivations for any type for the term.

We build both of these concepts into the type signature of the type checker, as seen in Figure 5. The definition uses Agda's `Dec` type for its return value: a value of type `Dec X` is either a `yes` containing a value of `X`, or a `no`, containing a proof that no `X` exists. In this case, the `Dec` contains a dependent tuple[3] of the type checker's outputs, and a typing derivation relating them to the inputs. Soundness is guaranteed by the attached typing derivation, while completeness is guaranteed by the fact that `Dec` requires a proof in case of failure.

Similarly to the type checker's output, terms themselves carry some proofs with them, most notably about the correctness of their scoping. Terms are parametrised over a scope α (as seen

[2]Strictly speaking, this process could be called type inference. However, that term usually refers to more advanced inference algorithms.

[3]A dependent tuple is a tuple in which the type of each value can depend on the previous values.

```
TypecheckResult : TypingContext → Term α → Set
TypecheckResult Γ t = Dec $ Σ[ T ∈ Type ]
                            Σ[ Γ' ∈ TypingContext ]
                            Σ[ Ω ∈ TypingContext ]
                            (Γ ⊢ t :: T , Γ' ⨟ Ω)

typeOf : (Γ : TypingContext) → (t : Term α) → TypecheckResult Γ t
```

Fig. 5. The type signature of the type checker

```
data Term (α : Scope) : Set where
  `_#_ : (x : name) → x ∈ α → Term α
  `split_as_,_⇒_ : Term α → (x y : name) → Term (α ‣ x ‣ y)
    → {x ∉ α} → {y ∉ α} → {x ≢ y}
    → Term α
  - etc.
```

Fig. 6. The definition of variable and split terms. Variable terms carry a proof of well-scopedness ($x \in \alpha$). Split terms carry proofs of freshness for their newly bound variables (*e.g.* $x \notin \alpha$), as well as a proof of their distinctness. The scope of the subterm is also extended with the new variables (`Term (α ‣ x ‣ y)`).

on Figure 5), which is a list of variable names that are in scope for the term. This is used to enforce two properties: firstly any referenced variables must be in scope, and secondly, all new variables must use distinct names from previously defined variables. An example of such proofs is shown in Figure 6. The first property ensures the well-scopedness of all variables, while the second property ensures that there are never two variables with the same name in scope. This uniqueness of variable names is useful because it allows the typing rules to completely ignore issues such as name shadowing.

While many other programs that process terms use de Bruijn indices [6], we do not. The reason for this is that our typing rules frequently remove existing bindings from typing contexts, which would be challenging to express using de Bruijn indices.

Embedding proofs in terms simplifies type checking and ensures that ill-scoped terms cannot accidentally be constructed anywhere. However, these proofs must also be generated somewhere: we do so in the *scope checker*.

The scope checker takes as input a raw term, that is, a term without any proofs attached, and outputs either a term annotated with proofs, or an error if the term is ill-scoped. In the process, the scope checker renames all variables to unique names.

3 IMPLEMENTATION

We implement the type checker described previously in Agda version 2.6.4.3, with standard library version 2.0. The entire implementation comprises approximately 1300 lines of code, available online.[4]

Section 3.1 describes the Agda encoding of λ_{sub} types and terms. Section 3.2 describes the implementation of the relations necessary for the type checker. Section 3.3 describes the implementation of the `typeOf` function. Finally, Section 3.4 describes the implementation of the scope checker.

Throughout this section, and the entire type checker, we make frequent use of Agda's mixfix operators to define the syntax of various objects immediately when defining the object. As an example, the identifier `_∪_≡_` defines a ternary operator, with the underscores marking the place

[4] https://github.com/CakeWithSteak/suty

```
data Type' : Set
data Pretype : Set

data Pretype where
  Bool : Pretype
  Unit : Pretype
  _`×_ : Type' → Type' → Pretype
  _⇒_ : Type' → Type' → Pretype

data Type' where
  `_`_ : Qualifier → Pretype → Type'
```

Fig. 7. A definition of types that is elegant, but permits invalid types (ordered functions)

```
data Type : Set where
  `_`Bool : Qualifier → Type
  `_`Unit : Qualifier → Type
  `_`_`×_ : Qualifier → Type → Type → Type
  `_`_⇒_ : (q : Qualifier) → Type → Type → {q ≢ ord} → Type
```

Fig. 8. The actual definition of types used in the type checker

of the operands. The defined operator can then be applied as follows: $\Omega_1 \cup \Omega_2 \equiv \Omega_3$. This allows us to keep the code close to the notation defined in Section 2.

The entire type checker can be compiled to Haskell via Agda's MAlonzo backend. The GitHub repository includes a set of examples against which the type checker will run as a demonstration.

3.1 Defining types, terms and qualifiers

Types and terms are perhaps the most fundamental elements of a type system – however, in λ_{sub}'s case, there are some unique difficulties with their definitions in Agda, most notably in the case of types. These difficulties stem from the fact that ordered functions are not permitted.

If ordered functions were allowed, the definition shown in Figure 7 would suffice. The definition follows Walker [15] in first defining unqualified "pretypes", and then defining types as qualified pretypes. This definition is natural and easy to use, as it allows for simple pattern matching on qualifiers. However, it also permits ordered functions, which are not valid in λ_{sub}.

We instead adopt the definition shown in Figure 8. This definition embeds qualifiers directly into types, and forbids ordered functions by requiring a proof that the qualifier used with function types is not `ord`.

This definition is less amenable to pattern matching, as one cannot extract only the qualifier while ignoring the rest of the type. To alleviate this, we introduce a helper function, `qualifierOf`, which extracts the qualifier from a type. This is important for variable typing rules, which each require the variable type to have a specific qualifier: for instance, the Agda definition of TUVar takes an argument of type qualifierOf $T \equiv$ un. Unfortunately, constructing these proofs is not straightforward: one can pattern match on the output of `qualifierOf`, but Agda is not always able to unify the necessary expressions. An example is shown in Figure 9: the hole (`{! refl !}`) in Figure 9b cannot be refined, as Agda is unable to conclude that (qualifierOf T) $\equiv$ un. To solve this issue, we introduce another helper function, `qualifierCases`, which takes as input a type, and a function to handle each of the five possible qualifiers. Each function takes as input a type of qualifierOf $T \equiv$ q, and returns the same output type. The function is implemented by pattern matching on all possible type/qualifier combinations. An example of usage is shown in Figure 9c.

```
data Goal : Set where
  goal : (T : Type) → qualifierOf T ≡ un → Goal
```

(a)

```
f : (T : Type) → Maybe Goal
f T with qualifierOf T
... | un = just (goal T {! refl !})
... | _ = nothing
```

(b)

```
f' : (T : Type) → Maybe Goal
f' T = qualifierCases T
    (λ is-un → just (goal T is-un)) n n n n
  where
    n : ∀ {q} → qualifierOf T ≡ q → Maybe Goal
    n = const nothing
```

(c)

Fig. 9. An example of difficulties with unification in Agda. The goal is to write a function of type `Type → Maybe Goal`, with `Goal` defined in (a). (b) tries to use `qualifierOf`, and fails while (c) instead uses `qualifierCases`.

An alternative to forbidding ordered functions in the definition of types would be to forbid them via either the typing rules or some preprocessing step and adopt the simpler definition shown in Figure 7. While this would simplify certain parts of the type checker's implementation, we believe that it would overall make the system more complex and less reliable. If enforced at a preprocessing stage, the type checker itself would no longer work if called with inputs that come from elsewhere than the preprocessor, which could introduce bugs. If enforced in the typing rules, the rules would become significantly more complicated. A further advantage of ensuring that ordered functions cannot be represented is that should the project be extended with a compiler or interpreter that manipulates types, these extensions would be intrinsically protected from invalid types.

Unlike for types, the definition of terms is straightforward and follows the example given in Figure 6. Aside from the scoping-related proofs mentioned previously, lambda abstractions also carry a proof that their qualifier is not `ord`, forbidding the creation of ordered functions.

This presents another alternative to forbidding ordered function types: we could allow their types to be represented, but forbid creating them. In such a system, any type of the form $\mathsf{ord}\ T_1 \to T_2$ is valid but uninhabited, thus one can write terms such as $\mathsf{un}\ \lambda f : (\mathsf{ord}\ (\mathsf{un}\ \mathsf{Unit}) \to (\mathsf{un}\ \mathsf{Unit})).\ t$, where t is some term. This system is sound but confusing: it is not obvious that the argument of the function above is uninhabited, and thus the function can never be called. To avoid such confusion, we forbid ordered function types as described above.

3.2 Implementing relations

The typing relation depends on a few auxiliary relations, as defined in Section 2.2. The most significant of these relation are inclusion ($x \in \alpha$), ordered inclusion ($(x \mapsto T) \in^* \Gamma$), binding deletion ($\Gamma - (x \mapsto T) = \Gamma'$), context division ($\Gamma_1 ; \Omega \div \Gamma_2 = \Gamma_3$), context union and intersection ($\Omega_1 \cup \Omega_2 = \Omega_3$, $\Gamma_1 \cap_\mathsf{q} \Gamma_2 = \Gamma_3$), qualifier ordering ($q_1 \sqsubseteq q_2$), and containment relations ($q \langle T \rangle$, $q \langle\!\langle \Gamma \rangle\!\rangle$).

Each of these relations needs two elements: a definition, which is an Agda data type that represents proofs of the relation, and an inference algorithm, that given some subset of the relation's domains, derives the remaining domains, or potentially concludes that there are no values that can complete the relation.

We will use the context union relation as an example. Its definition and inference algorithm are presented in Figure 10. Contexts themselves are defined like standard cons lists, with the empty list written $\varnothing$, and appending to the context written as $\Gamma, x \mapsto T$. Proofs for context unions are defined as follows: the union of a context with an empty context is the context itself (`empty` constructor), while the non-empty case is defined inductively in the `append` constructor.

```
data _∪_≡_ {V : Set} : Context V → Context V → Context V → Set where
  empty : ∀ {Ω} → Ω ∪ ∅ ≡ Ω
  append : ∀ {Ω₁ Ω₂ Ω₃ x v} → Ω₁ ∪ Ω₂ ≡ Ω₃ → Ω₁ ∪ (Ω₂ , x ↦ v) ≡ (Ω₃ , x ↦ v)

mergeContext : (Ω₁ Ω₂ : Context V) → Σ[ Ω₃ ∈ Context V ] Ω₁ ∪ Ω₂ ≡ Ω₃
mergeContext Ω₁ ∅ = Ω₁ , empty
mergeContext Ω₁ (Ω₂ , x) with mergeContext Ω₁ Ω₂
... | Ω₃ , Ω₃-proof = (Ω₃ , x) , append Ω₃-proof
```

Fig. 10. The definition and inference algorithm of the context union relation

The context union is evaluated by the `mergeContext` function, which given two contexts, returns their union and a proof of this union. This makes the inference algorithm correct-by-construction, and it is also inherently complete, as any two contexts have a union.

Proofs and inference algorithms for other relations are defined similarly. One notable difference is that some inference algorithms do not always succeed, for instance, affine context intersection can fail if the two contexts contain different non-affine bindings. Inference algorithms for such relations return a `Dec`, ensuring completeness by returning a proof of impossibility in case of failure.

The typing relation is itself defined like these auxiliary relations, with its inference algorithm being the type checker itself, as shown in Figure 5. Typing rules are converted into constructors of the Agda type representing the typing relation using the auxiliary relations defined above. A few examples are shown in Figure 11. Note that, as some terms contain embedded proofs, the relevant typing rules (*e.g.* TUVar) need to take these proofs as parameters to be able to construct the necessary terms in their conclusions.

3.3 Implementing the type checker

Having defined the necessary relations, we now move to implementing the type checker itself. The path taken by well-typed programs is very similar to that of usual type checkers, with the exception that correctness proofs of intermediate results need to be assembled into the final proof of well-typedness using the typing rules.

Most of the complexity in the type checker itself is to account for ill-typed programs. In an ordinary type checker, if some condition for type correctness fails, *e.g.* the type of an if condition is not Boolean, it suffices to reject the program with some error message. However, in order to achieve completeness of our type checker, we need to return a proof of ill-typedness in case of failure.

In order to prove ill-typedness in Agda, we demonstrate that should any type be assigned to the term in question, a contradiction would follow. As an example, consider the case when the condition of an if term is not a Boolean. Assume, for the sake of contradiction, that the term is actually well-typed: in that case, there must be a valid typing derivation for it. The last rule of this typing derivation must be TIf, since no other rules are applicable to if terms. TIf requires that the condition of the if term be a Boolean, but we have established that this is not the case: therefore we have a contradiction.

Note that the proof above relies on the implicit assumption that each term has only one type: in other words, in the typing relation, the type, output context, and usage context are functionally dependent on the input context and the term. For brevity, we will refer to functional dependencies between the "input" values and the "output" values as the uniqueness of the relation. The definition of the typing relation does not encode this fact – we must prove it manually.

Proving the uniqueness of the typing relation requires proving that the relations it uses are also unique. This leads to a total of five uniqueness lemmas, listed in Figure 12.

```
data _⊢_::_,_⨟_ (Γᵢ : TypingContext) :
  (t : Term α)
  (ty : Type)
  (Γₒ : TypingContext)
  (Ω : TypingContext) → Set where
  TUVar : (p : x ∈ α)
        → (x ↦ T ∈* Γᵢ)
        → (qualifierOf T ≡ un)
        ──────────────────────
        → Γᵢ ⊢ (‘ x # p) :: T , Γᵢ ⨟ ∅
  TOVar : (p : x ∈ α)
        → (x ↦ T ∈* Γᵢ)
        → (qualifierOf T ≡ ord)
        → (Γₒ : TypingContext)
        ──────────────────────
        → Γᵢ - x ↦ T ≡ Γₒ → Γᵢ ⊢ (‘ x # p) :: T , Γₒ ⨟ ∅
  TRVar : (p : x ∈ α)
        → (x ↦ T ∈* Γᵢ)
        → (qualifierOf T ≡ rel)
        ──────────────────────
        → Γᵢ ⊢ (‘ x # p) :: T , Γᵢ ⨟ (∅ , x ↦ T)

  TIf : Γᵢ ⊢ t₁ :: ‘ q ‘Bool , Γ₂ ⨟ Ω₁
        → Γ₂ ⊢ t₂ :: T , Γ₃ ⨟ Ω₂
        → Γ₂ ⊢ t₃ :: T , Γ₄ ⨟ Ω₃
        → Γ₃ ∩ₐ Γ₄ ≡ Γ₅
        → Ω₂ ∩ᵣ Ω₃ ≡ Ω₄
        → Ω₁ ∪ Ω₄ ≡ Ω₅
        ──────────────────────
        → Γᵢ ⊢ ‘if t₁ then t₂ else t₃ :: T , Γ₅ ⨟ Ω₅
  TLet : (p : x ∉ α)
        → Γᵢ ⊢ t₁ :: T₁ , Γ₂ ⨟ Ω₁
        → Γ₂ , x ↦ T₁ ⊢ t₂ :: T₂ , Γ₃ ⨟ Ω₂
        → Γ₃ ⨟ Ω₂ ÷ ∅ , x ↦ T₁ ≡ Γ₄
        → Ω₁ ∪ Ω₂ ≡ Ω₃
        ──────────────────────
        → Γᵢ ⊢ (‘let x := t₁ ⇒ t₂) {p} :: T₂ , Γ₄ ⨟ Ω₃
    - etc.
```

Fig. 11. Agda encoding of the typing rules shown in Figure 3

```
∈*-unique : x ↦ T ∈* Γ → x ↦ U ∈* Γ → T ≡ U
÷-unique : Γ₁ ⨟ Ω ÷ Γ₂ ≡ Γ₃ → Γ₁ ⨟ Ω ÷ Γ₂ ≡ Γ₄ → Γ₃ ≡ Γ₄
contextIntersection-unique : Γ₁ ∩[ q ] Γ₂ ≡ Γ → Γ₁ ∩[ q ] Γ₂ ≡ Γ' → Γ ≡ Γ'
mergeContext-unique : Ω₁ ∪ Ω₂ ≡ Ω₃ → Ω₁ ∪ Ω₂ ≡ Ω₃' → Ω₃ ≡ Ω₃'
typing-unique : Γ ⊢ t :: T₁ , Γ'₁ ⨟ Ω₁ → Γ ⊢ t :: T₂ , Γ'₂ ⨟ Ω₂ → T₁ ≡ T₂ × Γ'₁ ≡ Γ'₂ × Ω₁ ≡ Ω₂
```

Fig. 12. Type signatures of the uniqueness lemmas. All variables are implicitly universally quantified.

```
data ScopeCheckResult {α : Scope} (t : RawTerm) : Set where
  good : Σ[ t' ∈ Term α ] EraseTerm t' t → ScopeCheckResult t
  bad : String → ScopeCheckResult t

scopeCheck : (t : RawTerm) → ScopeCheckResult {∅} t
```

Fig. 13. The type signature of the scope checker

3.4 The scope checker

The scope checker acts as a preprocessing stage for the type checker, enriching raw terms with proofs ensuring, among other, its well-scopedness and renaming variables to ensure uniqueness of variable names. The type of the scope checker is shown in Figure 13.

To ensure the soundness of the scope checker, we define the notion of erasure: an enriched term t' erases to a raw term t (denoted $EraseTerm\, t'\, t$) if converting t' to a raw term by deleting all its embedded proofs and renaming variables to their original names yields t. When producing an enriched term from a raw term, the scope checker returns a proof that the enriched term erases to the raw term, ensuring that the scope checker cannot change the meaning of the term.

Unlike the type checker, the scope checker is not proven complete, instead, it returns an error message on failure. This is because proving completeness would require producing a proof that no enriched term erases to the input raw term and producing such proofs is difficult as it requires reasoning about α-equivalent terms.[5] As the scope checker is only a preprocessing step for the actual type checker, which is the focus of this project, we leave this issue to future work.

4 DISCUSSION

We have described a correct-by-construction type checker for a substructural type system. In this section, we reflect on the merits of such a type checker, and whether it is useful in practice.

The raison d'être of a correct-by-construction type checker is to prevent bugs: it is therefore natural to ask whether bugs in type checkers actually exist and whether they are a problem. The answer is that bugs in type checkers are quite common: Chaliasos *et al.* [4] study typing-related bugs in popular JVM languages, and found a total of 4,153 such bugs. Their analysis of a representative sample of those bugs found that these bugs take a median of 24 days and a mean of 186 days to be fixed and that the three most prevalent types of bugs are "unexpected compile-time errors" (50.09%), "internal compiler errors" (24.69%) and "unexpected runtime behavior" (16.56%).

In light of the prevalence of type checker bugs, correct-by-construction type checkers have a strong promise of soundness and completeness, and might also reduce the amount of effort spent on writing tests. However, they also have an associated cost in that correct-by-construction type checkers are significantly more time-consuming to create than conventional type checkers.

The vast majority of the work in writing the type checker presented in this paper has been spent not on the type checking code itself, but rather on the infrastructure around it, namely the relations described in Section 3.1. This process is quite different from what most developers are used to, as it requires thinking on both the definition and the implementation level at the same time: one Agda encoding of a given concept may be more elegant, but harder to implement than the alternatives.

Programming using dependent types is also challenging in that the metalanguage may have limitations that are not immediately obvious to the programmer. In particular, as illustrated by the example of `qualifierOf` in Section 3.1, the compiler is not always able to unify more complex expressions. Since this limitation is not apparent at the time of defining a potentially problematic interface, this can lead to programmers accidentally creating abstractions that are difficult to use, leading to additional development time when the abstraction needs to be adjusted or replaced entirely.

[5] Two terms are considered α-equivalent if they differ only on bound variable names.

Another factor complicating correct-by-construction type checker development is that extending and refactoring them is inherently more difficult than for conventional type checkers. This is because many proofs and lemmas depend on manually handling all possible variants of some data type, thus any modification of the data type results in a large number of broken proofs that need to be manually updated. This issue is exacerbated by the fact that dependently typed programming languages are currently considered to be niche, and, as a result, there are far fewer tools available for automated refactoring than for mainstream languages.

The development cost of correct-by-construction type checkers is sufficiently high that we consider them to be infeasible for mainstream general-purpose programming languages. Such languages are updated relatively frequently – making the difficulty of extending the type checker an issue – and have user bases so large that, together with existing testing methods, the probability of a critical bug remaining in the type checker is very low.

However, in certain cases, the correctness of the type checker is of special importance: proof assistants need users to trust that the proofs they accept are really correct and in safety-critical cases even extremely rare bugs could be catastrophic. As such, we argue that for such languages, correct-by-construction programming can, and should, be used to ensure that the type checker is correct.

Finally, note that correct-by-construction programming guarantees only that the type checker follows the typing rules of the language – it does not provide any guarantees that these typing rules themselves do what their writer intended. Soundness bugs may still arise from incorrect typing rules, thus any desired property of the type system should be proven in addition to using correct-by-construction type checkers.

5 RESPONSIBLE RESEARCH

The results of this research can be reproduced by downloading the code available online,[6] and compiling it using the appropriate Agda version. We believe that there are no direct or reasonably foreseeable indirect ethical implications of this research.

6 RELATED WORK

There are a number of prior results on verified type checkers. Casamento [3] presents a framework for constructing correct-by-construction type checkers in Agda, specifically focusing on scoping. Sozeau *et al.* [11] present a verified type checker for the Coq proof assistant, itself written in Coq. Finally, Tan *et al.* [12] formally specify the type system of CakeML (a subset of Standard ML), prove its soundness, and develop a verified type inferencer, all in the HOL4 proof assistant.

Correct-by-construction programming can also be used for interpreters, as seen in the work of Bach Poulsen *et al.*, [2] which presents methods for writing intrinsically-typed interpreters in Agda.

More broadly, there is significant interest in verified compilers. Other than the verified type checker mentioned earlier, CakeML [8] features an entire verified compiler. Similarly, CompCert [9] is a verified compiler for the C programming language, targeting critical embedded systems.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a correct-by-construction type checker for a toy language with substructural types. We use Agda's dependent type system to define the toy language's type system and ensure that the type checker is sound and complete with respect to the definition. This means that the type checker provably accepts programs if and only if they are well-typed. We discuss that the development of such type checkers is more time-consuming than that of conventional type checkers, and therefore argue that conventional type checkers are likely more suitable for general-purpose programming languages than correct-by-construction type checkers. However, we

[6] https://github.com/CakeWithSteak/suty

believe that correct-by-construction type checkers do have an application within proof assistants and safety critical applications, where the impact of type checker bugs is especially high.

Importantly, correct-by-construction type checkers do not guarantee anything about the correctness of the typing rules themselves. As such, the typing rules must also be proven correct with respect to the properties they are meant to uphold.

Consequently, the most interesting area of future work would be to extend the type checker into a full correct-by-construction interpreter for the toy language. This would make it possible to prove that the guarantees that substructural type systems promise are indeed upheld by the typing rules, therefore increasing confidence in the language as a whole.

REFERENCES

[1] Agda Developers. [n. d.]. *Agda*. https://agda.readthedocs.io/

[2] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–34. https://doi.org/10.1145/3158104

[3] Katherine Casamento. 2019. *Correct-by-Construction Typechecking with Scope Graphs*. Technical Report. https://doi.org/10.15760/etd.7145

[4] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (oct 2021), 30 pages. https://doi.org/10.1145/3485500

[5] Jawahar Chirimar, Carl A Gunter, and Jon G Riecke. 1996. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming* 6, 2 (1996), 195–244.

[6] N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[7] Atsushi Igarashi and Naoki Kobayashi. 2000. Garbage collection based on a linear type system. In *Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC'00)*, Vol. 152.

[8] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

[9] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not.* 41, 1 (jan 2006), 42–54. https://doi.org/10.1145/1111320.1111042

[10] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. 2003. A type theory for memory allocation and data layout. *ACM SIGPLAN Notices* 38, 1 (2003), 172–184.

[11] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–28. https://doi.org/10.1145/3371076

[12] Yong Kiam Tan, Scott Owens, and Ramana Kumar. 2015. A verified type system for CakeML. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages* (Koblenz, Germany) *(IFL '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/2897336.2897344

[13] Jesse A Tov and Riccardo Pucella. 2011. Practical affine types. *ACM SIGPLAN Notices* 46, 1 (2011), 447–458.

[14] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.

[15] David Walker. 2004. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, Chapter 1, 30–36.