Correct-by-Construction Implementation of Typecheckers

Typechecking records with depth and width subtyping

Kazimierz Ciaś

Supervisor(s): Jesper Cockx, Sára Juhošová

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to the EEMCS Faculty of 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: Kazimierz Ciaś

Final project course: CSE3000 Research Project

Thesis committee: Jesper Cockx, Sára Juhošová, Thomas Durieux

Abstract

Typecheckers help avoid bugs in code by catching errors early. Their implementation can, however, be incorrect, leading to inconsistencies in their operation. This research explores how we can use Agda and correct-by-construction programming to create a typechecker guaranteed to be correct in its implementation. For this purpose, I based a toy language on the simply typed lambda calculus extended with records and subtyping. The resulting typechecker is proven to be sound and complete with respect to the typing and subtyping rules of the toy language. This paper compares the correct-by-construction method to existing typecheckers. The new approach offers a greater degree of trust in its implementation but comes at the cost of being more demanding to develop and maintain.

1. Introduction

Statically typed programming languages allow developers to create programs confidently by eliminating many possible errors, such as passing a `String` value to a function expecting an `Int`. In opposition to dynamic type systems, which check the compatibility of types during execution, static type safety checks, performed by the so-called typecheckers, verify correctness without running the code, thus catching even unlikely possibilities leading to errors at the expense of some flexibility and convenience [1]. They also enable immediate feedback to the programmer as they compose their code, shortening the time between writing it and catching bugs.

Unfortunately, typecheckers are susceptible to bugs themselves, causing them to be inconsistent with their specification and even, in some cases, unsound—meaning they will approve an incorrectly typed program. Luckily, there are ways to minimise the possibility of making errors. One such technique is correctness-by-construction, an approach to programming in which the program starts as a small set of rules and is then incrementally refined to achieve the desired result without risking critical mistakes. Together with the Curry-Howard correspondence, a paradigm describing how types are a direct analogy to logical propositions [2], we can obtain rigorous mathematical proofs that our typecheckers are correct. We can use Agda—a programming language with built-in support for Curry-Howard correspondence and correctness-by-construction [3]—to create a typechecker with certainty that it will adhere to the

formal description of the type system.

This report demonstrates how Agda, together with the principles of Curry-Howard correspondence and correct-by-construction programming, can be used to create a typechecker for the simply typed lambda calculus [4] extended with record types and subtyping. Section 2 explores the concepts forming the foundation of this research. In Section 3, I elaborate on the challenges encountered by existing typechecker and describe the formal definition of the type system I implemented. I explain how it is transcribed into Agda and explain the details of my implementation in Section 4, and in Section 5, I compare the result to other approaches to typechecking, taking into account the reliability and trustworthiness as well as ease of development and possibility for further expansion. Section 6 acknowledges previous research done in the area of correct-by-construction typechecking. Lastly, Section 7 touches upon the reproducibility of this research, and Section 8 summarises the findings and identifies open avenues for potential future research.

2. Background

2.1. Typecheckers

The development of static typecheckers has been vital in ensuring the reliability of modern programming languages. Many widely used languages—such as Rust, Typescript and Haskell—employ static typechecking, illustrating the practical application of theoretical type systems.

Agda Agda is both a functional programming language and a proof assistant. Its typechecker supports dependent types, enabling the expression of complex properties and the construction of proofs within the type system. Agda is used for writing programs where correctness is critical, as it allows developers to encode and verify program properties directly in the type system. Agda's typechecker is highly expressive, providing powerful tools for formal verification and theorem proving.

2.2. Subtyping

Cardelli and Wegner introduced foundational concepts of subtyping relations in type systems in 1985 [5]. Their research built the base for understanding how programming languages can be made more flexible with subtyping by allowing programmers to assign values whose type does

not directly equal the type expected by, for example, the function argument, but rather any value with a compatible type, reducing the need to create exhaustive conversion functions. In record types, subtyping arises in two forms: depth subtyping—where the type of a field in a record is replaced with its subtype—and width subtyping—which adds more fields to a record.

3. Formal specification

The first step to creating a typechecker is to lay out a mathematical model of the type system it describes. This is a well-researched topic, so I opted to adapt the algorithmic typing rules laid out by Pierce [1] instead of constructing them from scratch.

This chapter discusses the most relevant typing rules of the implemented system. For the sake of brevity, this paper omits the syntax and typing rules of the simply typed lambda calculus, only focusing on record types and subtyping. The complete syntax and all rules can be found in Appendix A.

3.1. Records

Extending the base of the simply typed lambda calculus, I adopted the rules for constructing records and retrieving values from them as proposed by Pierce [1]:

$$\frac{\text{for each } i \qquad \Gamma \vdash \mathrm{t}_i : \mathrm{T}_i}{\Gamma \vdash \{l_i = {t_i}^{i \in 1..n}\} : \{l_i : {T_i}^{i \in 1..n}\}} \text{ [record]}$$

$$\frac{\Gamma \vdash t : \{l_i : {T_i}^{i \in 1..n}\}}{\Gamma \vdash t.l_j : T_j} \text{ [projection]}$$

Here, [record] aggregates multiple terms into a labelled collection, i.e. a record, and [projection] allows for retrieving a value with a given label from a record. To make the implementation of the [record] rule in Agda easier, I split it into two rules: [record-empty] as the base case, creating an empty record, and [record-more], which, given a record, adds to it one more field.

$$\frac{}{\Gamma \vdash \{\} : \{\}} \text{ [record-empty]}$$

$$\frac{\Gamma \vdash \{l_i = {t_i}^{i \in 1..n}\} : \{l_i : {T_i}^{i \in 1..n}\} \qquad \Gamma \vdash t_{n+1} : T_{n+1}}{\Gamma \vdash \{l_i = {t_i}^{i \in 1..n+1}\} : \{l_i : {T_i}^{i \in 1..n+1}\}} \text{ [record-more]}$$

3.2. Subtyping

With the above rules, wherever the typechecker expects a certain type, it will only accept a provided term if its type exactly matches the expected one. This, however, limits the developer. The type system could instead allow any term whose type is compatible with the expected type. Such compatibility is called subtyping and is denoted as $S <: T$, meaning that S is a subtype of T.

To justify the existence of subtyping, I have introduced three new types: naturals, integers and the top type.

The top type (denoted here as $\top$) has no members—no constant term has type $\top$—but it is the universal supertype—all types are subtypes of $\top$:

$$\forall T \quad T <: \top \text{ [sub-top]}$$

Subtyping is reflexive; otherwise, we would not be able to pass the exact expected type. This characteristic could be represented as $\forall T\ T <: T$, but that would overlap with [sub-top] and the subtyping rules for functions and records, which would make the proofs of one type not being a subtype of another more complicated. I instead split it into two separate rules for naturals and integers:

$$\mathbb{N} <: \mathbb{N} \text{ [sub-natural]} \qquad\qquad \mathbb{Z} <: \mathbb{Z} \text{ [sub-integer]}$$

Finally, a natural number can be used wherever an integer is expected, thus:

$$\mathbb{N} <: \mathbb{Z} \text{ [sub-nat-int]}$$

For functions, if we take two types, $S = S_1 \to S_2$ and $T = T_1 \to T_2$, for $S <: T$ to hold, there are two prerequisites we need to fulfil. First, we must be able to pass to S all arguments that can be passed to T, so T_1 must be a subtype of S_1. Then, the return type of S must be a subtype of the return type of T.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \text{ [sub-arrow]}$$

Lastly, there are three ways of creating subtypes within record types. First, since the order of fields in records is not functionally significant, a record is equivalent to its permutation, so it is its subtype. There, we replace the type of a field with its subtype. We can also create a subtype by adding an entirely new field in the method called width subtyping.

The above methods, especially permutations, would be challenging to implement directly. We can combine them all into one subtyping rule stating that the set of labels in the original record must be a subset of the set of labels in the subtype—combining permutations and width subtyping—and that all fields which exist in both records must themselves, in the record subtype, be subtypes of the corresponding fields in the expected record type. Somewhat counterintuitively, despite the rule seemingly having more responsibilities, it is easier to implement in Agda, which I explain in more detail in section 4.

$$\frac{\{l_i^{i\in 1..n}\} \subseteq \{k_j^{j\in 1..m}\} \qquad k_j = l_i \implies S_j <: T_i}{\{k_j : S_j^{\,j\in 1..m}\} <: \{l_i : T_i^{\,i\in 1..n}\}} \text{ [sub-record]}$$

Four rules in my type system require a provided term to have a particular type. Three of those regard constructing naturals and integers, but those require a term of type $\mathbb{N}$, which does not have any subtypes except for itself, so it does not require modification. The last is the rule for function application, which needs to be changed to allow the argument to be a subtype of the function's parameter type.

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_{11} \to T_{12} \\ \Gamma \vdash t_2 : T_2 \qquad T_2 <: T_{11}\end{array}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ [application]}$$

Without subtyping, function application would require the argument t_2 to have the same type as the parameter of the function t_1—T_{11}. Instead, the [application] rule allows T_2 to be a subtype of T_{11}, giving the developer greater freedom by not demanding them to write conversion functions where such conversion is trivial.

4. Implementation

Creating a correct-by-construction typechecker in Agda is a very linear process consisting of two main steps: first, transcribing the formal model of the type system—the variable contexts, syntax, and typing rules—and second, writing the typechecking and inference functions. This chapter presents an overview of this procedure, touching upon Agda's quirks but omitting less relevant details for brevity. The entire code of my implementation is available on GitHub[1]. The general steps in this chapter are adaptable for creating typecheckers with different features to the one presented here .

[1] https://github.com/silentstormm/cbc-type-checker

I started my implementation of the typechecker from a basic typechecker for the simply typed lambda calculus provided as an example by the supervisors[2].

4.1. Type system

The first step in creating the typechecker was transcribing the type system from the mathematical model into Agda, starting with defining the data structure for variable contexts.

```
1 Map : Set → Set → Set
2 Map K V = List (K × V)
3
4 _∈′_ : ∀ {K V} → K × V → Map K V → Set
5 x ∈′ m = First (x ≢_) (x ≡_) m
6
7 _∈ₖ′_ : ∀ {K V} → K → Map K V → Set
8 k ∈ₖ′ m = First ((k ≢_) ∘ proj₁) ((k ≡ₖ_) ∘ proj₁) m
```

I store bound variables in a list of pairs, where the first value is the identifier and the second is its associated type. I also use `Map`s for record types and terms.

I defined two membership relations based on `First` from Agda's standard library, which is a dependently typed predicate on lists, stating that a predicate (in this case inequality) holds for each element of the list the type is indexed on until an element for which another predicate (here equality) is true. The first is a standard list membership—Agda provides a list membership based on `Any`, but it isn't suitable here, as I allow for variable shadowing. This means that when retrieving a variable which has been bound multiple times, I always need to retrieve the latest binding. Note that the `K` and `V` parameters are in curly braces, marking them as implicit—Agda can infer them from the value of `K × V`. Additionally, `∀` denotes that the rule holds for all possible `K` and `V`, and signals to Agda that it can infer their type. From this point, I will refer to this membership as the list membership.

The second relation, which I call the "key" membership, states that a key exists in the map regardless of its assigned value, which can be used with a lookup function to retrieve the value. This way the key membership can be upgraded into the list membership.

The next pieces of the type system are its terms and types:

[2] https://github.com/sarajuhosova/cbc-type-checker

```
data Type : Set where
  `⊤ : Type
  `ℕ : Type
  `ℤ : Type
  _⟹_ : Type → Type → Type
  Rec : Map name Type → Type

data Term : Set where
  `_ : name → Term
  ƛ_::_⟹_ : name → Type → Term → Term
  _·_   : Term → Term → Term
  rec : Map name Term → Term
  get : name → Term → Term
  `zero : Term
  `suc : Term → Term
  `pos : Term → Term
  `negsuc : Term → Term
```

`Type` describes three base types: the top type, naturals, and integers; the function type and the record type, while `Term` defines the term of my language. The function abstraction term `ƛ_::_⟹_` is a mixfix operator; in Agda, underscores declare places for arguments in the definition of a function or constructor. The term takes the type of its parameter as an argument, as otherwise, type inference would require implementing algorithms well outside the scope of this research. `name` is a parameter of the modules in which `Type` and `Term` are defined, which lets me change the type of variable identifiers in one place and have it propagate to all files in the project.

The last remaining parts of the type system are its typing and subtyping rules. Those are also encoded as data types, as they can be viewed as the axioms of the type system.

```
data _⊢_::_ (Γ : Map name Type) : Term → Type → Set where
  ...

data _<:_ : Rel Type 0ℓ where
  ...
```

The typing rule type is dependent on the value of a context, as the validity of a typing rule can depend on the contents of the context—specifically, this is true for the rule for variables:

```
⊢`
  : ∀ {x} {A} (p : (x , A) ∈′ Γ) (q : x ∈ₖ′ Γ)
  → A ≡ lookup′ q
  → Γ ⊢ ` x :: A
```

Here, the rule is valid only if the variable is bound within its context. The rule went through several iterations before I found a version that satisfies Agda's limitations. At first, it only required the key membership:

```
⊢`
  : ∀ {x} (p : x ∈ₖ′ Γ)
  → Γ ⊢ ` x :: lookup′ p
```

This approach, however, is flawed. Because the resulting type contains a function (`lookup′ p`), Agda gets stuck during unification and constraint solving involving this rule. That causes it to be unable to verify the correctness of proofs output by the typechecker when it rejects a program. To solve this, I added the list membership, which provides the type bound to the variable. The third parameter (`A ≡ lookup′ q`) ensures that `p` points to the latest binding.

Two other rules are notable in their notation in Agda—record expansion ([rec-more]) and projection:

```
⊢rec-more
  : ∀ {x} {v} {A} {rs : Map name Term} {rt : Map name Type}
  → ¬ (x ∈ₖ′ rt)
  → Γ ⊢ rec rs :: Rec rt
  → Γ ⊢ v :: A
  → Γ ⊢ rec ((x , v) :: rs) :: Rec ((x , A) :: rt)

⊢get
  : ∀ {x} {A} {v} {r : Map name Term} (p : (x , A) ∈′ r) (q : x ∈ₖ′ r)
  → A ≡ lookup′ q
```

```
    → Γ ⊢ v :: Rec r
    → Γ ⊢ get x v :: A

```

Since fields cannot be added or removed from a record after its creation and their types cannot be modified, `⊢rec-more` requires a proof that the new label `x` does not yet exist in the record.

The notation of `⊢get` is analogous to that of `⊢`` as their operation is very similar—variable contexts and record types are both collections associating identifiers with types.

Of subtyping rules, all are denoted exactly as in Section 3.2, with the exception of the one pertaining to records:

```
<:rec
  : ∀ {m n}
  → n ⊆′ m
  → All (λ { (k , v) → (i : k ∈ₖ′ m) → lookup′ i <: v}) n
  → Rec m <: Rec n
```

The first argument is a $\in_k'$-based proof that the set of labels in `n` is a subset of labels in `m`.

The second argument is another predicate on lists from Agda's standard library. It states that for every element `(k , v)` of the map `n` if the label `k` also exists in `m`, its corresponding type in `m` is a subtype of `v`. Thus, it is equivalent to the proposition $k_j = l_i \implies S_j <: T_i$ from the [sub-record] rule in the specification.

4.2. Type checking and inference

After having transcribed the type system into Agda, the next step was to devise an interface checking whether a program is well-typed. For that purpose, I created three functions:

```
_<:?_ : ∀ {S T} → Dec (S <: T)

infer : ∀ (Γ : Map name Type) u → Dec (Σ[ t ∈ Type ] Γ ⊢ u :: t)

```

```
check
  : ∀ (Γ : Map name Type) u (t : Type) → Dec (Σ[ s ∈ Type ] (s <: t × Γ ⊢ u :: s))
```

Breaking it down, `Dec` is a predicate from Agda's standard library, which states that a proposition is decidable. It is constructed with a boolean value saying whether the proposition is true and a proof reflecting said value. Thus `S <:? T` decides whether S is a subtype of T, returning the appropriate subtyping rule or a proof that it is not. `infer` takes a context `Γ` and term `u`, and decides a dependent pair `(Σ[ t ∈ Type ] Γ ⊢ u :: t)`, where the value of the first element `t` of type `Type` determines the type of the second element. This forces the function to return a typing rule that is relevant to the returned type—for example, it can't return `(`ℤ , ⊢zero)`, as `⊢zero` derives the type to be `` `ℕ ``. `check` builds on the two previous functions—it infers the type of `u`, checks whether it is a subtype of the given type `t`, and returns the inferred type and appropriate typing and subtyping rules if successful.

`_<:?_` does not pass Agda's termination check because of the recursive `lookup′ i <: v` call within the `Rec m <:? Rec n` case, as it is not structural recursion, which is the only type of recursion allowed by Agda. In reality, the call does not make the function non-terminating, as `lookup′ i` is an element of `m`, and thus always strictly smaller. Because of that, the function is annotated with the `{-# TERMINATING #-}` pragma, which tells Agda to treat the function as terminating and ignore any termination check errors.

The typechecker is guaranteed to be sound because every program it approves can be derived from the typing and subtyping rules and complete because every program derived from the typing rules is approved, or, by contraposition, no program it rejects can be derived from the rules. This follows from the fact the typechecker returns the relevant proof when it accepts or rejects a program.

5. Results

It is no secret that typecheckers can contain bugs. At the time of writing, the Typescript repository had over 1500 bugs[3], many related to typechecking. Rust has 88 soundness holes, called "the worst kind of bug" by the description of the issue label[4]. These bugs would not be possible with correct-by-construction programming, at least assuming the language used to cre-

[3] https://github.com/microsoft/TypeScript/issues?q=is%3Aopen+is%3Aissue+label%3ABug
[4] https://github.com/rust-lang/rust/issues?q=is%3Aopen+is%3Aissue+label%3AI-unsound

ate the typechecker is itself sound. These assurances make correct-by-construction typecheckers especially useful in high-stakes scenarios, such as when creating spacecraft software or software handling sensitive user data.

On the other hand, correct-by-construction programming does not come without downsides. Because of Agda's restrictions, it is often necessary to prove seemingly trivial propositions. Take, for example, the proposition `∀ Γ term t u → Γ ⊢ term :: t → Γ ⊢ term :: u → u ≡ t`, saying that any given term has only one possible type under a particular context. This is immediately apparent from the typing rules—there is only one possible derivation for any given term. However, Agda does not see that requiring the developer to write auxiliary lemmas which would not have been necessary in other programming languages. Another example where Agda fails to find the solution by itself occurs when deciding whether two types are equal:

```
1 _≟_ : DecidableEquality Type
2 `⊤ ≟ `⊤ = yes refl
3 `⊤ ≟ a = no λ ()
```

Agda throws an error on the second case, stating that `` `⊤ `` could still be equal to `a`, despite the fact that the only case where equality is possible is already covered. Only if I split the second case into separate cases for each remaining type does Agda approve the trivial proofs of inequality. This forces the developer to create $O(n^2)$ cases with n being the number of constructors in the compared datatype, where $O(n)$ cases should have been sufficient. The most recent version of my typechecker has 682 lines of code[5]; 129 of those—over a sixth—deal with such trivial cases, making the code harder to read and understand. If I could merge the cases as in the listing above, this number would be reduced by over six times to, at most, 19 lines.

Another point of common frustration with Agda is bugs in its proof assistance capabilities. Quite often, when using the case splitting and automatic proof search features, Agda produces code which does not compile—sometimes the problem is minor, like incorrectly qualifying imported entities, but at times Agda successfully splits cases or fills a proof hole only to complain about getting stuck when trying to verify the correctness of its solution.

Correct-by-construction programming presents a unique set of challenges which add

[5]Including empty lines

complexity to the development process, creating a trade-off between reliable verification and the rate of advancements. This effect is apparent in the history of this project—creating the typechecker discussed in this paper took 41 days between my first and last commit to the Git repository while a version with no assurances of soundness—also written in Agda, available on the `unsound` branch of the repository[6]—took merely 2 hours. It's important to note that this was my first project using Agda; however, even though someone with previous experience would have likely been more efficient, I believe there would still be a disparity in the time those two approaches take.

6. Related work

Sozeau, Boulier, Forster, *et al.* [6] implement a correct-by-construction typechecker for another proof assistant, Coq. Their solution is also written in Coq, allowing it to verify its own correctness. The result has much more features compared to my typechecker, but it does not implement record subtyping. Tan, Myreen, Kumar, *et al.* [7] describe the compiler of CakeML, a subset of ML implemented and verified in the HOL4 theorem prover [8]. The research showcases the use of correct-by-construction programming for creating an entire compiled language, ensuring its correctness at each from parsing to the final machine code while Tan, Owens, and Kumar [9] focus on CakeML's type system. Notably, the language does not support records. There are also multiple more specialised languages built on top of CakeML, including the systems programming language Pancake [10], and PureCake, a lazy functional language similar to Haskell [11].

7. Responsible Research

The code of the typechecker, along with instructions for running, modifying and verification, is available publicly on GitHub[7] under the MIT open-source license.

The data used to compare the result of this research and other typecheckers is available publicly in their respective repositories under various open-source licenses.

[6] https://github.com/silentstormm/cbc-type-checker/tree/unsound
[7] https://github.com/silentstormm/cbc-type-checker

8. Conclusion

8.1. Results

This research presents a way of creating a typechecker using the principles of correct-by-construction programming in Agda. Adapting the algorithmic typing rules laid out in Pierce [1], I constructed a typechecker which, when deciding whether a program is well-typed, returns a proof of its judgement being consistent with the typing rules, therefore ensuring its correctness. Comparing the result to more conventional ways of creating a typechecker reveals that correct-by-construction programming increases the trustworthiness of a program, making it especially suitable for critical systems where failure can have a high impact. However, if the technique were to be viable for general use, the tools around it need improvement. Even then, correct-by-construction code is often more complex than code without the assurances of correctness, resulting in significantly slower development.

8.2. Future work

Scalability While this paper demonstrates the feasibility of creating a correct-by-construction typechecker in Agda, I have not investigated the scalability of this approach. As such, it's unclear how much its complexity would increase when adding more features.

Performance Another avenue to explore is the performance of correct-by-construction typechecking compared to other approaches. Many IDEs, whether through native language support or the Language Server Protocol [12], provide immediate feedback including type errors on typing, without the need to start compilation manually. This means the typecheckers are expected to run quickly to minimise latency and consume little resources, as checks are performed quite often.

Agda developer experience Lastly, Agda has some issues increasing the complexity of using it in practical applications. Research into enhancing Agda's unification and constraint-solving algorithms could provide ways to alleviate those concerns. Features which would improve the ease and speed of development in Agda but might not require extensive research include code completion with automatic importing and better documentation of the standard library.

References

[1] B. C. Pierce, *Types and programming languages*, English. Cambridge, Mass.: MIT Press, 2002. [Online]. Available: http://www.books24x7.com/marc.asp?isbn=0262162091.

[2] W. A. Howard, "The formulae-as-types notion of construction," in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980, pp. 479–490, ISBN: 9780123490506. [Online]. Available: https://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf.

[3] U. Norell 1979., "Towards a practical programming language based on dependent type theory," Chalmers University of Technology, Göteborg, 2007.

[4] A. Church, "A formulation of the simple theory of types," *The Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940, ISSN: 00224812. [Online]. Available: http://www.jstor.org/stable/2266170 (visited on 04/28/2024).

[5] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Comput. Surv.*, vol. 17, no. 4, pp. 471–523, Dec. 1985, ISSN: 0360-0300. DOI: 10.1145/6041.6042. [Online]. Available: https://doi.org/10.1145/6041.6042.

[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," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. DOI: 10.1145/3371076. [Online]. Available: https://doi.org/10.1145/3371076.

[7] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified CakeML compiler backend," *Journal of Functional Programming*, vol. 29, 2019. DOI: 10.1017/S0956796818000229. [Online]. Available: jfp19.pdf.

[8] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *Principles of Programming Languages (POPL)*, ACM Press, Jan. 2014, pp. 179–191. DOI: 10.1145/2535838.2535841. [Online]. Available: https://cakeml.org/popl14.pdf.

[9] Y. K. Tan, S. Owens, and R. Kumar, "A verified type system for CakeML," in *Implementation and Application of Functional Programming Languages (IFL)*, Awarded the Peter Landin prize for best paper, ACM Press, 2015. DOI: 10.1145/2897336.2897344. [Online]. Available: https://cakeml.org/ifl15.pdf.

[10] J. Å. Pohjola, H. T. Syeda, M. Tanaka, *et al.*, "Pancake: Verified systems programming made sweeter," in *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, ser. PLOS '23, New York, NY, USA: Association for Computing Machinery, 2023, pp. 1–9. DOI: 10.1145/3623759.3624544. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3623759.3624544.

[11] H. Kanabar, S. Vivien, O. Abrahamsson, *et al.*, "PureCake: A verified compiler for a lazy functional language," in *Programming Language Design and Implementation (PLDI)*, ACM, 2023. [Online]. Available: pldi23-purecake.pdf.

[12] N. Gunasinghe and N. Marcus, *Language Server Protocol and Implementation*. Apress, Jan. 2022. DOI: 10.1007/978-1-4842-7792-8.

A. Full formal specification

A.1. Terms

$\langle term \rangle$ ::=	x	variable
	$\lambda x : \langle type \rangle . \langle term \rangle$	function abstraction
	$\langle term \rangle\ \langle term \rangle$	function application
	zero	zero
	suc $\langle term \rangle$	successor
	pos $\langle term \rangle$	positive integer
	negsuc $\langle term \rangle$	negative successor
	$\{\langle label \rangle : \langle term \rangle, ...\}$	record
	$\langle term \rangle . \langle label \rangle$	projection

A.2. Types

$\langle type \rangle$ ::=	$\top$	top
	$\mathbb{N}$	natural
	$\mathbb{Z}$	integer
	$\langle type \rangle \to \langle type \rangle$	function
	$\{\langle label \rangle : \langle type \rangle, ...\}$	record

A.3. Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ [\text{variable}]$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}\ [\text{abstraction}]$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 <: T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ [\text{application}]$$

$$\frac{}{\Gamma \vdash \text{zero} : \mathbb{N}}\ [\text{zero}]$$

$$\frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \text{suc}\ t : \mathbb{N}}\ [\text{suc}]$$

$$\frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \text{pos}\ t : \mathbb{Z}}\ [\text{pos}]$$

$$\frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \text{negsuc}\ t : \mathbb{Z}}\ [\text{negsuc}]$$

$$\frac{}{\Gamma \vdash \{\} : \{\}}\ [\text{record-empty}]$$

$$\frac{\Gamma \vdash \{l_i = t_i{}^{i \in 1..n}\} : \{l_i : T_i{}^{i \in 1..n}\} \qquad \Gamma \vdash t_{n+1} : T_{n+1}}{\Gamma \vdash \{l_i = t_i{}^{i \in 1..n+1}\} : \{l_i : T_i{}^{i \in 1..n+1}\}}\ [\text{record-more}]$$

$$\frac{\Gamma \vdash t : \{l_i : T_i{}^{i \in 1..n}\}}{\Gamma \vdash t.l_j : T_j}\ [\text{projection}]$$

A.4. Subtyping rules

$$T <: \top\ [\text{sub-top}]$$

$$\mathbb{N} <: \mathbb{N}\ [\text{sub-natural}]$$

$$\mathbb{Z} <: \mathbb{Z}\ [\text{sub-integer}]$$

$$\mathbb{N} <: \mathbb{Z}\ [\text{sub-nat-int}]$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}\ [\text{sub-arrow}]$$

$$\frac{\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \qquad k_j = l_i \implies S_j <: T_i}{\{k_j : S_j{}^{j \in 1..m}\} <: \{l_i : T_i{}^{i \in 1..n}\}}\ [\text{sub-record}]$$