Building a Correct-by-Construction Type Checker for a Dependently Typed Core Language

Bohdan Liesnikov[1[0009−0000−2216−8830]] and Jesper Cockx[1[0000−0003−3862−4073]]

Delft University of Technology, Delft, the Netherlands
{b.liesnikov, j.g.h.cockx}@tudelft.nl
bohdan.liesnikov.name, jesper.cx

Abstract. Dependently typed languages allow us to state a program's expected properties and automatically check that they are satisfied at compile time. Yet the implementations of these languages are themselves just software, so can we really trust them? The goal of this paper is to develop a lightweight technique to improve their trustworthiness by giving a formal specification of the typing rules and intrinsically verifying the type checker with respect to these rules. Concretely, we apply this technique to a subset of Agda's internal language, implemented in Agda. Our development relies on erasure annotations to separate the specification from the runtime of the type checker. We provide guidelines for making design decisions for certified core type checkers and evaluate trade-offs.

Keywords: Dependent types · Agda · Correct-by-Construction Programming.

1 Introduction

Developers use a variety of techniques to increase trust in the software projects they are working on, ranging from manual testing, to static type systems and formal specification and verification. The latter can guarantee adherence of the software to the specification - as demonstrated by projects such as CompCert [33], CakeML [31], sel4 [30], JSCert [13], and Verdi Raft [55]. The correctness of these formal verification efforts relies on the soundness of the tools used [8, 41] - such as Coq [51], Agda [50], Idris [14, 52], or Isabelle [37].

Being mere pieces of software, formal verification tools can also have bugs and are not inherently trustworthy. To mitigate this, a common countermeasure is to build them around a small and trusted kernel, as pioneered by LCF [28], Coq, and Twelf [39], and later adopted in Lean, Isabelle [32], Idris, and others. Andromeda developers [11] describe the kernel as "kept as simple as possible, and it only supports very straightforward type-theoretic constructions which directly correspond to applications of inference rules and admissible rules." While this does increase trust, they also note that "a careful code review of the nucleus will probably unearth some bugs, and hopefully not very many." The same can be said about all other proof assistants, as witnessed by critical bugs that are still being discovered[1], even if these are hard to exploit accidentally [46].

[1] github.com/coq/coq/blob/master/dev/doc/critical-bugs.md

Moving beyond a trusted core, MetaCoq [44, 46] proposes to formally verify each part of the verification pipeline - from parsing to extraction. However, for dependently typed languages this can be a herculean task, taking teams many years to complete. MetaCoq itself began in 2014 with TemplateCoq [34] and is still ongoing in 2023 [46]. Thus existing approaches fail at either cost-effectiveness or strong verification assurances. We need techniques that provide a more rigorous verification process than pure code review, yet remain scalable and feasible for real-world sized systems.

In this paper we present a design that sits between these two extremes. We target a dependently typed core language modelled after a subset of Agda's internal syntax. Concretely, we contribute the following:

- We formally specify the syntax and typing rules for a dependently typed language with universes, dependent function types, simple datatypes, and case expressions.
- We implement a type checker for this language that produces evidence of well-typedness for each term it accepts.
- We demonstrate the use of erasure annotations [9, 24, 36, 50] to ensure a clear separation between the parts of the type checker that are needed for computation, and those only needed for its verification.

The implementation consists of four parts: a well-scoped representation of the syntax for terms and signatures (Sect. 2, Sect. 4.1, and Sect. 5.1), a simple environment machine for reduction (Sect. 3.3, Sect. 4.2, and Sect. 5.2), a formal specification of the typing and conversion judgments (Sect. 3.1, 4.3, and 5.3), and a correct-by-construction type checker that outputs typing derivations (Sect. 3.2, 4.4, and 5.4).

While the language we present is far from novel [10, 21], the focus of this paper is on how we formalize the syntax and typing rules, and how these choices influence the implementation of the type checker.

We introduce our implementation gradually, starting with the simply typed lambda calculus (STLC) in Sect. 2 and Sect. 3. In Sect. 4 we extend it to handle dependent function types and universes. Finally, in Sect. 5 we add simple inductive datatypes and defined symbols.

The source code for the paper is available at github.com/jespercockx/agda-core/tree/aplas-2024.[2]

Limitations This paper is an experiment in language engineering rather than language theory. While we provide a formal specification of the syntax and typing rules, we refrain from proving any meta-theoretical properties. In particular, we are not formalising a variant of MLTT but take the typing rules as the source of truth. For the type checker we aim to strike a balance between formal guarantees and resources requried, in particular we prove soundness of our type checker but not completeness, as doing so would require inversion lemmas for our typing judgments. We also do not check termination or positivity, and hence do not ensure logical soundness.

[2] archived at doi.org/10.4121/6f239149-2526-42a0-8d07-d0e9d6714f7f

2 Representing well-scoped syntax

In this section, we present a well-scoped syntax for STLC, which relies on an abstract interface for representing scopes and names.

2.1 Well-scoped syntax for STLC

Well-scoped [3, 12] syntax representations capture the variable names that can be used within a term. For STLC we define a type of terms parameterised by an abstract scope α. Since the scope parameter is marked as erased (`@0`), at runtime the representation is equivalent to the plain Haskell datatype `Term` on the right.

```
data Term (@0 α : Scope name) : Set where
  TVar  :  (@0 x : name) → x ∈ α
        → Term α
  TLam :  (@0 x : name) → (v : Term (x ◃ α))
        → Term α
  TApp  :  (u : Term α) → (v : Term α)
        → Term α
```

```haskell
data Term where
  TVar  :: Int
        -> Term
  TLam  :: Term
        -> Term
  TApp  :: Term -> Term
        -> Term
```

Variables (TVar) consist of an erased name together with a proof of inclusion $x \in \alpha$, i.e. that x is in scope α. At runtime, this proof corresponds to a plain de Bruijn index of type `Int`. The TLam constructor binds x and ensures that the body of the lambda v is in a larger scope $x \triangleleft \alpha$, where α is the current ambient scope.

Types in STLC are defined as a simple datatype with a base type TyNat and a function type TyArr. Since there are no type variables, no scope is needed.

```
data Type : Set where
  TyNat : Type
  TyArr : (a b : Type) → Type
```

2.2 Scopes and their operations

To represent variables, language specifications have to choose between named variables - which are easy for humans to read but hard to reason about - and de Bruijn indices or other nameless representations - which are easier to formalize but notoriously confusing to humans. Our representation combines the best of both worlds by representing variables as an erased name together with a proof that it is in scope, which compiles to a de Bruijn index.

While scopes could be represented as a simple list of names, we choose to work with an abstract interface [29]. This allows us to switch to a more efficient representation if needed, and be explicit about which operations on scopes we require. This will prove to be useful later on, as we use scopes to model not just local variables but also global definitions, which can be much more numerous. Concretely, the interface we rely on is as follows:

– A type Scope : Set with constructors $\emptyset$ for empty scopes, [_] for singleton scopes, and _<>_ for the disjoint union of two scopes.

– An operator $\sim$ for reversing the order of the variables in a scope.
– A subscope predicate $\_\subseteq\_$: @0 Scope → Scope → Set, with operations for deciding equality of subscope witnesses and computing the smaller scope's complement. A membership predicate $\_\in\_$: @0 name → @0 Scope → Set is defined as $x \in \alpha = [\ x\ ] \subseteq \alpha$.
– A data structure All : (p : @0 name → Set) → @0 Scope → Set storing an element of type $p\ x$ for each name x in the scope, with a lookup operation.

The name argument to the singleton constructor [_] is erased, meaning that the names provide the extra convenience for writing Agda, but do not have any impact on the runtime representation of scopes. We do not enforce uniqueness of names, but the inclusions are unique and their equality is decidable.

As we will see later, certain definitions require a runtime representation of the scope. For this purpose, we use the type Rezz $a\ x$ of resurrections [24] of an erased variable @0 x : a, which contains a non-erased value that is propositionally equal to x. For example, we will later require a weakening function that converts Term β to Term (α <> β). In terms of de Bruijn indices it needs the size of α and Rezz Scope α is precisely that - a runtime representation of the spine, not to the names contained within it. Generally speaking, we can resurrect a value if we can recompute it - for example, we can resurrect the scope from the context - see rezzScope in Sect. 3.1 for usage. This is because the context is indexed by the scope, so the length of the context is precisely the size of the scope, which is Rezz Scope α.

Discussion Using well-scoped syntax helps us spot mistakes in the specification of our language - which matters since many bugs in type systems arise from incorrect handling of variables.

Our choice to use an abstract type of names rather than a simpler type of well-scoped de Bruijn indices is motivated by keeping our specification as readable as possible. It also gives more informative types to syntax operations, and rules out certain classes of errors that will be easier to miss with plain de Bruijn indices. For example, a function of type Term ($x \triangleleft y \triangleleft s$) → Term ($y \triangleleft x \triangleleft s$) makes it clear that the order of the variables x and y is swapped, while the type Term ($2 + s$) → Term ($2 + s$) does not tell us anything about the order.

Going beyond well-scoped syntax, one might also argue in favour of well-typed syntax, statically ruling out even more errors. However, defining well-typed syntax for languages with type-level computation is notoriously tricky [5, 6, 19], and would force us to define syntax and typing judgment in a mutually dependent way. In addition, it would not free us from also having to define untyped syntax, since we cannot assume the input to our type checker to be well-typed.

Finally, since the type checker expects the input to be well-scoped, we fundamentally rely on the parser to perform scope-checking. While in principle it is possible to perform scope and type checking in a single pass, we choose to keep them separate, thus gaining modularity but requiring the parser to be verified separately, which we defer to future work.

3 Type checking STLC

Now that we have a syntax for STLC, let us take a look at three other big pieces: the specification of typing rules, the type checker, and the evaluator.

3.1 Typing rules

To specify the typing rules of STLC, we need to define a type of contexts which will store the types of variables. The type of contexts is indexed by the scope of variables declared. We define Γ , $x : t$ as syntactic sugar for CtxExtend Γ x t.

```
data Context : @0 Scope name → Set where
  CtxEmpty : Context ∅
  CtxExtend : Context α → (@0 x : name) → Type → Context (x ◃ α)
```

The typing judgment TyTerm (rendered as $\Gamma \vdash u : t$) is indexed by a context, a term, and its type. Each constructor of TyTerm corresponds to a typing rule.

```
data TyTerm (@0 Γ : Context α) : @0 Term α → @0 Type → Set where
  TyTVar :   (p : x ∈ α)
             → Γ ⊢ TVar x p : lookupVar Γ x p
  TyLam :   Γ , x : a ⊢ u : b
           → Γ ⊢ TLam x u : TyArr a b
  TyApp :   Γ ⊢ u : (TyArr a b)
           → Γ ⊢ v : a
           → Γ ⊢ TApp u v : b
```

For variables, the type is given by the context. A lambda has a function type, with the body living in an extended context. Finally, application asserts the type of the argument matches the domain of the head symbol and the result matches the codomain.

We choose to state the rules in a declarative way since they serve as part of the specification and should be easily understood. However, in the implementation of the type checker below, we follow a bidirectional discipline [26, 40].

3.2 Type checking

To implement a certified type checker, we first define a simple type checking monad with a failure capability (tcError):

```
TCM : Set → Set
TCM a = Either TCError a
```

Type checking function application requires *conversion checking*, that is - checking whether two types are equal. Since STLC has no type-level computation, conversion is just syntactic equality, so the conversion checker returns a proof of equality or throws an error.

```
convert : (a b : Type) → TCM (a ≡ b)
convert TyNat TyNat = return refl
convert (TyArr a1 b1) (TyArr a2 b2) = do
```

```
    refl ← convert a1 a2
    refl ← convert b1 b2
    return refl
convert _ _ = tcError "unequal types"
```

We use Agda's do-notation for the TCM monad, which includes the ability to pattern match on the result of a statement. Here we match the results of the recursive calls against refl, unifying the left- and right-hand sides of the equality for the remainder of the do-block.

The type checker itself follows a bidirectional style, with two functions checkType and inferType that are defined mutually. Both functions return a typing derivation, where inferType also returns the type of the given term, while checkType checks it against a specific type.

```
inferType  : ∀ (Γ : Context α) u → TCM (Σ[ ty ∈ Type ] (Γ ⊢ u : ty))
checkType : ∀ (Γ : Context α) u (ty : Type) → TCM (Γ ⊢ u : ty)

inferType ctx (TVar x p)   = return (lookupVar ctx x p , TyTVar p)
inferType ctx (TLam x te) = tcError "cannot infer type of lambda"
inferType ctx (TApp u v)   = do
  (TyArr a b) , gtu ← inferType ctx u
    where _ → tcError "application head should have a function type"
  gtv ← checkType ctx v a
  return (b , TyApp gtu gtv)
```

We infer types for all terms, except for a lambda. The `where` clause in the clause for `TApp` deals with any cases that are not on the 'happy path' where the result of the recursive call is a `TyArr`. In checking mode we type check only lambdas and for any other term we switch modes and perform a conversion check.

```
checkType ctx (TLam x v) (TyArr a b) = do
  gtv ← checkType (ctx , x : a) v b
  return (TyLam gtv)
checkType ctx (TLam x v) _ =
  tcError "lambda should have a function type"
checkType ctx u ty = do
  (gtu , dgty) ← inferType ctx u
  refl ← convert gtu ty
  return dgty
```

3.3 Reduction

While it is not yet necessary for the type checker, we also implement an evaluator for terms, as we will need it for checking dependent types. It is based on a call-by-value Krivine machine [22, 43].

First, we define environments as lists of terms where each term can refer to the previous ones. They are indexed by an initial scope α and a final scope β.

```
data Environment : (@0 α β : Scope name) → Set where
  EnvNil   :   Environment α α
```

```
  EnvCons :   Environment α β → (@0 x : name) → Term β
           → Environment α (x ◃ β)
```

The state of the evaluator consists of an environment, the current term it is focused on, and a stack of arguments it still needs to apply this term to. Both the focus and the stack can refer to values defined in the environment.

```
record State (@0 α : Scope name) : Set where
  constructor MkState
  field
    @0 {fullScope} : Scope name
    env   : Environment α fullScope
    focus : Term fullScope
    stack : List (Term fullScope)
```

The machine itself takes one step of reduction at a time, using the step function. Just s means that another reduction step is possible and Nothing means that the evaluation is done.

```
step : (s : State α) → Maybe (State α)
step (MkState e (TVar x p) s) = case lookupEnvironment e p of λ where
  (Left _)  → Nothing
  (Right v) → Just (MkState e v s)
step (MkState e (TApp v w) s) = Just (MkState e v (w :: s))
step (MkState e (TLam x v) (w :: s)) =
  Just (MkState (e , x ↦ w) v (map weakenBind s))
step (MkState e (TLam x v) []) = Nothing
```

Variables are looked up in the context, application arguments are pushed to the stack, and lambdas move arguments from the stack to the environment before continuing to evaluate the body.

We start evaluation with an empty environment and an empty stack. When the machine halts we still have to extract the reduced term from the final state. For this, we convert the environment to a substitution to apply it to the focus.

Substitutions Subst α β (syntactic sugar $\alpha \Rightarrow \beta$) are a list-like data structure indexed over two scopes α and β.

```
data Subst : (@0 α β : Scope name) → Set where
  SNil  : Subst ∅ β
  SCons : Term β → Subst α β → Subst (x ◃ α) β
```

The function substTerm (not shown here) takes a substitution $\alpha \Rightarrow \beta$, and applies it to a term in Term α to get a term in Term β.

Using substitution, we can convert the state back to a plain term:

```
unState : Rezz _ α → State α → Term α
unState r (MkState e v s) = substTerm (envToSubst r e) (applys v s)
```

Since step can be applied to ill-typed terms, repeated application does not necessarily terminate. Hence to define a multi-step reduction, we use a fuel argument of type Nat that indicates a maximum number of reduction steps.

```
reduceState : Rezz _ α → (s : State α) → Nat → Maybe (Term α)
reduceState r s zero = Nothing
reduceState r s (suc fuel) = case (step s) of λ where
  (Just s') → reduceState r s' fuel
  Nothing  → Just (unState r s)

reduce : Rezz _ α → (v : Term α) → Nat → Maybe (Term α)
reduce {α = α} r v = reduceState r (makeState v)
```

Discussion Using substitution to extract a term from the final state duplicates the terms present in the environment if they occur more than once in the result. To avoid this, we could add let-expressions to our language and use them to maintain the environments generated by the machine. However, a naive implementation of this approach introduces let-bindings for unused terms (user-defined, as well as generated from the machine's state) which in practice renders it unusable. We could remove these spurious lets with a garbage-collection-like procedure, but such a procedure would add extra complexity, and with it extra opportunities for bugs. So while the duplication caused by substitution is an annoying downside, it leads to more manageable terms in the output.

An alternative to using fuel is the Delay monad [1, 23]. In practice we ran into complications when trying to implement it: using Delay as part of a monad stack requires lifting other monads through it, which requires an altered definition to lift through later. To define Delay as a monad transformer we also need to ensure that m it transforms is strictly positive - either through a container encoding or implementing a new extension of Agda.

4 Dependent function types and universes

In this section, we extend STLC defined in Sect. 2 and Sect. 3 with dependent function types (Π-types) and a universe hierarchy, thus getting a minimal dependently typed language.

The main change in the syntax is that types can now contain variables and hence also have to be scoped. Type conversion also becomes more complicated, as it needs to reduce terms.

4.1 Syntax

As types now contain terms, they are now also indexed over a scope. Concretely, we represent types as a pair of a term together with its sort. These sorts can be inserted by the elaborator if needed.

```
data Sort α where
  STyp : Nat → Sort α
```

```
record Type α where
  inductive; constructor El
  field typeSort : Sort α
        unType   : Term α
```

One further change is that we wrap the argument to function application in the Elim datatype, which will prove useful for future extensions (Sect. 5).

```
data Term α where
  TVar  : (@0 x : name) → x ∈ α → Term α
  TLam  : (@0 x : name) (v : Term (x ◃ α)) → Term α
  TApp  : (u : Term α) (es : Elim α) → Term α
  TPi   : (@0 x : name) (u : Type α) (v : Type (x ◃ α)) → Term α
  TSort : Sort α → Term α

data Elim α where
  EArg : Term α → Elim α
```

Contexts are the same as before, except with types now also being well-scoped.

```
data Context : @0 Scope name → Set where
  CtxEmpty : Context ∅
  CtxExtend : Context α → (@0 x : name) → Type α → Context (x ◃ α)
```

4.2 Reduction

Evaluation of functions is the same as for STLC. Π-types and sorts do not reduce.

4.3 Typing and conversion rules

In this section, we extend the typing judgment with rules for Π-types and sorts. We also add a rule to convert a derivation between two types, which in turn requires the definition of a conversion judgment.

Since the context can be considered an ‘input’ to the typing judgment, our typing rules do not enforce well-formedness of the types in the context but instead assume it. However, they do enforce well-formedness of the type – as well as the term itself.

Typing judgments The form of the typing judgment is the same as before, apart from the added scope argument to the type.

```
data TyTerm (@0 Γ : Context α) : @0 Term α → @0 Type α → Set where
```

We omit the rule for variables since it is precisely the same as in Sect. 3. In the rule for TLam, the name of the variable in u is x while the variable in b is named y, so we need to rename the latter using renameTopType, which maps Type $(x \triangleleft \alpha)$ to Type $(y \triangleleft \alpha)$.

```
TyLam : Γ , x : a ⊢ u : renameTopType r b
        → Γ ⊢ TLam x u : El k (TPi y a b)

TyAppE : {b : Type α}
         → Γ ⊢ u : a
         → TyElim Γ u e a b
         → Γ ⊢ TApp u e : b
```

The application rule uses the auxiliary typing judgment TyElim, which checks that the head symbol is of Π-type and the argument type matches the domain. To get the type of the application it substitutes the argument into the codomain.

```
data TyElim (@0 Γ : Context α) :
  (@0 u : Term α) (@0 e : Elim α) (@0 t a : Type α) → Set where
    TyArg :   (unType c) ≅ TPi x a b
            → Γ ⊢ v : a
            → TyElim Γ u (EArg v) c (substTopType r v b)
```

For computing the sort of Π-types and sorts, we rely on two functions piSort (maximum) and sucSort (successor).

```
TyPi   :   Γ ⊢ u : sortType k
         → Γ , x : (El k u) ⊢ v : sortType l
         → Γ ⊢ TPi x (El k u) (El l v) : sortType (piSort k l)
TyType :   Γ ⊢ TSort k : sortType (sucSort k)
```

Finally, the conversion rule maps a typing derivation between convertible types.

```
TyConv : Γ ⊢ u : a → (unType a) ≅ (unType b) → Γ ⊢ u : b
```

Conversion rules We use an untyped conversion judgment Conv (syntactic sugar ≅), since it allows us to define conversion separately from typing. Once again, the rules themselves closely follow the literature[20, 38].

The two main conversion rules are CRedL and CRedR that allow us to reduce the left- and right-hand side respectively. These two rules use the predicate ReducesTo v w expressing that v reduces to w, when given sufficient fuel.

```
@0 ReducesTo : (v w : Term α) → Set
ReducesTo {α = α} v w = Σ[ (r , f) ∈ Rezz _ α × Nat ]
                              reduce r v f ≡ Just w
```

Aside from these two rules, conversion is reflexive and respects all term constructors.

```
data Conv where
  CRedL : @0 ReducesTo u u'        CApp : u ≅ u' → w ≃ w'
        → u' ≅ v → u ≅ v                 → TApp u w ≅ TApp u' w'
```

For Π-types and lambdas, we need to rename the variable on one side in order to bring both terms to the same scope.

```
CPi :   unType a ≅ unType a'                    CLam :   u ≅ renameTop r v
      → unType b ≅ renameTop r (unType b')             → TLam y u ≅ TLam z v
      → TPi x a b ≅ TPi y a' b'
```

Discussion Untyped conversion allows us to simplify conversion rules, but prevents us from easily adding type-directed conversion rules such as eta-expansion and proof irrelevance. Theoretically, it would be possible to ask for a typing derivation locally when applying these rules, but that would require conversion to at least maintain a typing context. Moreover, implementing a type checker that can provide these derivations would require a proof of subject reduction, which we chose not to develop.

This problem could be circumvented by adding a typing rule that axiomatises subject reduction. Since reduction is already part of the trusted code base, this does not further compromise soundness. However, it would complicate any future attempts to do metatheory.

4.4 Type checking and conversion checking

Conversion-checker The conversion checker has the following interface:

```
convert : ∀ Γ (t q : Term α) → TCM (t ≅ q)
```

Since checking conversion requires reduction, we extend the type checking monad TCM with a field storing a read-only fuel value. The top-level convert function gets this value and passes it to the auxiliary convertCheck, which recurses on it. The function reduceTo takes this fuel and a term v and returns the reduced term w together with a witness of type ReducesTo v w.

```
convertCheck : Nat → (r : Rezz _ α) → ∀ (t q : Term α) → TCM (t ≅ q)
convertCheck zero _ _ _ = tcError "need more fuel"
convertCheck (suc fl) r t q = do
  rgty ← reduceTo r t fl
  rcty ← reduceTo r q fl
  case (rgty , rcty) of λ where
```

To compare two variables, we use decidable equality of variable indices $x \in \alpha$. If the indices are equal, we match on refl to unify them so we can use CRefl.

```
(TVar x p ⟨ rpg ⟩ , TVar y q ⟨ rpc ⟩) →
  CRedL rpg <$> CRedR rpc <$>
    ifDec (decIn p q)
      (λ where {{refl}} → return CRefl)
      (tcError "two different variables aren't convertible")
```

Other terms are checked by a recursive descent. For example, for lambdas we check convertibility of the bodies, renaming variables as needed.

```
(TLam x u ⟨ rpg ⟩ , TLam y v ⟨ rpc ⟩) →
  CRedL rpg <$> CRedR rpc <$>
  CLam <$> convertCheck fl (rezzBind r) u (renameTop r v)
```

Type checker As before, we follow a bidirectional discipline, with only TyLam in checking mode again. When we encounter an inferrable term in a checkable position, we use the TyConv rule to switch modes.

```
checkCoerce : ∀ Γ (t : Term α) → Σ[ ty ∈ Type α ] Γ ⊢ t : ty
              → (cty : Type α) → TCM (Γ ⊢ t : cty)
checkCoerce ctx _ (ty , dty) cty =
  TyConv dty <$> convert ctx (unType ty) (unType cty)
```

We discuss two cases for illustrative purposes, the others are similar. To type check a lambda, we need to reduce the type before checking that it is a Π.

```
checkType ctx (TLam x u) (El s ty) = do
  let r = rezzScope ctx
  fuel ← tcmFuel
```

```
  (TPi y tu tv) ⟨ rtp ⟩ ← reduceTo r ty fuel
    where _ → tcError "couldn't reduce a term to a pi type"
  let cp = CRedR rtp CRefl
  d ← checkType (ctx , x : tu) u (renameTopType r tv)
  return $ TyConv (TyLam {k = s} d) cp
```

Type checking an application symbol relies on the auxiliary function inferElim.

```
inferType ctx (TApp u e) = do
  tu , gtu ← inferType ctx u
  a  , gte ← inferElim ctx u e tu
  return $ a , TyAppE gtu gte
```

The function inferElim itself again reduces the type to a Π-type and checks the argument against its domain.

```
inferElim ctx u (EArg v) tu = do
  let r = rezzScope ctx
  fuel ← tcmFuel
  (TPi x at rt) ⟨ rtp ⟩ ← reduceTo r (unType tu) fuel
    where _ → tcError "couldn't reduce head type to a pi type"
  gtv ← checkType ctx v at
  let tytype = substTopType r v rt
      gc     = CRedL rtp CRefl
  return $ tytype , TyArg gc gtv
```

5 Inductive types

In this section, we expand the language with parameterised - but not indexed - inductive types and case-expressions. We also add globally defined symbols. From an infrastructure point of view the main addition are a set of global scopes - defScope for global definitions, conScope for constructor names, and fieldScope for the fields of each constructor - and signature of global definitions (datatypes and symbols).

5.1 Syntax

There are two new constructors added to the syntax. TDef represents a global symbol, where the name d has to be in the global scope of definitions defScope. TCon is a fully applied datatype constructor - it takes the name of the constructor, an inclusion proof in the global scope of constructors, and a list of arguments (represented as a substitution).

```
data Term α where
  TDef : ∀ (@0 d) →    d ∈ defScope   → Term α
  TCon : ∀ (@0 c) (cp : c ∈ conScope) → (fieldsOf cp) ⇒ α → Term α
```

We also have a new constructor ECase for Elim, representing a case expression with a list of branches and a return type or motive [35], which can depend on the scrutinee. As this is a constructor of Elim, the scrutinee itself is implicit.

```
data Elim α where
  ECase : (bs : Branches α cs) (m : Type (x ◃ α)) → Elim α
```

Each branch matches on a specific constructor c. The scopes ensure that the term on the right-hand side can access all the arguments to the constructor. Since scopes are extended to the left but argument lists grow to the right, the order of the scope has to be inverted (∼).

```
data Branch α where
  BBranch :   (@0 c : name) (c∈cons : c ∈ conScope)
              (let args = fieldsOf c∈cons)
            → Rezz _ args → Term (∼ args <> α) → Branch α c
```

The type Branches α cs requires that there is one branch for each constructor in the scope cs, thus ensuring coverage.

```
data Branches α where
  BsNil  : Branches α ∅
  BsCons : Branch α c → Branches α cs → Branches α (c ◃ cs)
```

Signatures The scopes for defined symbols, constructors, and fields are collected in a type of Globals:

```
record Globals : Set where
  field defScope   : Scope name
        conScope   : Scope name
        fieldScope : All (λ _ → Scope name) conScope
```

The above provides only the *names* globally available, so we introduce another record Signature that associates a definition to each name in defScope: either a type and value for a global symbol, or a datatype declaration.

```
Signature : Set
Signature = All (λ _ → Type ∅ × Definition) defScope

data Definition where
  FunctionDef  : (funBody : Term ∅) → Definition
  DatatypeDef : (datatypeDef : Datatype) → Definition
```

Datatype declarations DatatypeDef store a sort dataSort, telescopes for parameters dataParTel, and a list of constructors dataConstructors. Both the telescopes and the list of constructors are also given an (erased) scope of the names they declare.

```
record Datatype where
  field @0 dataPars       : Scope name
        @0 dataCons       : Scope name
        dataSort          : Sort dataPars
        dataParTel        : Telescope ∅ dataPars
        dataConstructors : All (λ c → Σ (c ∈ conScope)
                                          (Constructor dataPars c))
                              dataCons
```

Each constructor definition stores a telescope for the types of its arguments. Since each part of the signature is guided by scopes, there is no risk of forgetting an argument when defining a datatype or its constructors.

```
record Constructor pars c cp where
  field conTelescope : Telescope pars (fieldsOf cp)
```

There are three well-formedness properties that we assume to hold:

1. From the information stored in the Datatype record we can compute its type, which should match the type given in the Signature.
2. The names of the constructors of each datatype should be distinct.
3. The types in each conTelescope should be no larger than dataSort.

5.2 Reduction

For the evaluator, we need new rules for unfolding defined symbols and for evaluating case expressions. For the latter, when the head symbol is reduced to a constructor, and the top element on the stack is a case elimination, we pick the appropriate branch, substituting the arguments into its body.

```
step sig (MkState e (TDef d q) s) = case getBody sig d q of λ where
  (Just v) → Just (MkState e (weakenGlobal v) s)
  Nothing → Nothing
step sig (MkState e (TCon c q vs) (ECase bs _ :: s)) =
  case lookupBranch bs c q of λ where
    (Just (r , v)) → Just $ MkState (extendEnvironment (revSubst vs) e)
                                    v
                                    (weakenRevEl r s)
    Nothing        → Nothing
```

5.3 Typing and conversion rules

Conversion We define three new conversion judgments for branches, lists of branches, and substitutions respectively.

```
data ConvBranch   {@0 α} : (@0 b₁  b₂  : Branch α cn)   → Set
data ConvBranches {@0 α} : (@0 bs₁ bs₂ : Branches α cs) → Set
data ConvSubst    {@0 α} : (@0 us₁ us₂ : β ⇒ α)         → Set
```

Constructors are convertible if they have the same name and convertible arguments.

```
data Conv {α} where
  CCon : (@0 cp : c ∈ conScope) {@0 us vs : fieldsOf cp ⇒ α}
       → ConvSubst us vs → TCon c cp us ≅ TCon c cp vs
```

Two case statements are convertible if their motives are convertible, and for each constructor the corresponding bodies are convertible.

```
CECase : (bs bp : Branches α cs)
         (ms : Type (x ◃ α)) (mp : Type (y ◃ α))
```

$\rightarrow$ renameTop $\{y = z\}$ r (unType ms) $\cong$ renameTop r (unType mp)
$\rightarrow$ ConvBranches bs bp $\rightarrow$ ECase bs ms $\simeq$ ECase bp mp

Global references TDef are convertible when the inclusions are equal, same as for TVar.

Typing A defined symbol has the type indicated in the signature *sig*.

```
data TyTerm {α} Γ where
  TyDef : (@0 p : f ∈ defScope)
         → Γ ⊢ TDef f p : weakenGlobalType (getType sig f p)
```

A constructor is well typed if its name c belongs to a datatype d and its arguments are typeable with respect to the telescope conTelescope of this constructor. The type of the constructor is computed by the constructorType function, which returns a type of the form TDef d dp.

```
TyCon :   ( @0 dp : d ∈ defScope) (@0 dt : Datatype)
        → ( @0 cq : c ∈ dataCons dt)
        →   @0 getDefinition sig d dp ≡ DatatypeDef dt
        → (let (cp , con) = lookupAll (dataConstructors dt) cq)
        → { @0 pars : dataPars dt ⇒ α}
        → { @0 us : fieldsOf cp ⇒ α}
        → (let ds = substSort pars (dataSort dt))
        → TySubst Γ us (substTelescope pars (conTelescope con))
        → Γ ⊢ TCon c cp us : constructorType d dp c cp con ds pars us
```

Substitutions are typed with respect to a telescope: each term in the substitution must be typeable with the corresponding type from the telescope.

A branch is well-typed if its body is well-typed with respect to a specialised motive. The motive is specialised to the constructor, which is applied to fresh variables from a context extended with the constructor arguments.

```
data TyBranch {α} Γ dt ps rt where
  TyBBranch : (c∈dcons : c ∈ dataCons dt)
    → (let (c∈cons , con ) = lookupAll (dataConstructors dt) c∈dcons
           ctel = ⋯; bsubst = ⋯ )
    → ∀ {rf} (rhs : Term (~ fieldsOf c∈cons <> α))
    → TyTerm (addContextTel ctel Γ) rhs (substType bsubst rt)
    → TyBranch Γ dt ps rt (BBranch c c∈cons rf rhs)
```

Finally, the TyBranches judgment simply checks well-typedness of each branch.

5.4 Type and conversion checker

Now that the typing rules are set, writing the type checker is mostly a mechanical task. Thus for brevity, we omit the actual definitions in this section and instead highlight the main challenges.

Representing type constructors Since we model type constructors as TDef we have to extract the parameters from it manually, via reduction to a TApp and a traversal of the eliminations in it. This is to ensure that the arguments to TDef are well-typed with respect to the declaration in the signature. Fundamentally, it is not a challenge, but in hindsight, having a dedicated constructor for type constructors would have made the implementation easier.

Coverage checking During the type checking of a TCase elimination we have to ensure that the Branches cover all constructors of the datatype. The type checker can tell us that the branches cover a certain scope β, we need to ensure that it matches dataCons dt, as required by the rule. To do this, we need to compare the run-time representation of both scopes, hence they need to be resurrected. We do it via the function allBranches : Branches α β $\rightarrow$ All (λ c $\rightarrow$ $c \in$ conScope) β, and the list of all constructors coming from the dataConstructors field of dt. Since both contain an inclusion proof $c \in$ conScope associated with each c, we can establish their (in-)equality. In cases like this we find it helpful to think about the runtime representation of any decision taken, with a positive decision supported by an erased proof.

Well-scoped substitutions Since names ensure a lot of important correspondences, working with *well-scoped substitutions* makes it much easier to see when a substitution can be applied to a term - or how it should be lifted to do so. It also eliminates corner cases where we know statically that the sizes of two scopes are the same, which is helpful during development and reduces the number of potential bugs.

6 Related Work

Minimising the trusted computing base (TCB) of type checkers and proof checkers is not a new idea. It originated with de Bruijn [15], but only recently it has become possible to formalise the specification of a real-world core language and a few of those have been done. Below, we list related works in the order of decreasing topic proximity.

MetaCoq [44, 45] is a formalization of Coq's core language in Coq. It originated from a formalisation by Barras [16] and a more recent metaprogramming development known as Template Coq [7, 34]. The main difference with MetaCoq is that we aim to develop a certified type checker with minimal metatheory, while MetaCoq wants to be a full formalization of the Coq core. Aside from that, we also make a few different design decisions. First, we rely on Agda's erasure annotations instead of the `Prop` universe in Coq. Second, we use a well-scoped representation instead of plain de Bruijn indices. Third, while MetaCoq also uses a Krivine machine for reduction but it has a separate specification of the reduction rules, so the MetaCoq evaluator is not part of the TCB, while ours is. Fourth, our typing judgments assume well-formedness of contexts rather than requiring a proof of it for every rule. Fifth, we formalise only the declarative style of typing rules, but follow MetaCoq in using a bidirectional style for the type checker.

Adjedj et al. [4] formalise meta-theory for MLTT with Π, Σ , natural numbers, and an `Id` type. They also develop a simple complete and correct type checker for this language. There are four big differences between our works. On the surface, as MetaCoq, they use plain de Bruijn indices with Coq definitions derived using AutoSubst [42, 47]. On a higher lever, the variant of MLTT they formalise uses recursors for the two induction types instead of pattern-matching. They also formalise typed reduction, with one of the main contributions being a reformulation of work by Abel et al.[2] to avoid induction-recursion. While we are interested in typed reduction, the complications they run into arise from meta-theoretical proofs. The techniques we develop are more light-weight since we do not do meta-theory. At last, their approach does not immediately allow extraction, while ours does, due to erasure.

Strub et al. [48] develop a self-certifying type checker for F*. This work requires a developed metatheory of the language in Coq, which we do not have. Many of the design decisions are similar to MetaCoq, so the differences mentioned there apply here too. They define reduction in terms of substitution, while we use a Krivine machine. Finally, we argue that due to focus on readability and simplicity, our design is overall less complicated and more compact.

Carneiro [17] develops a type checker ("external verifier") for Lean 4 in Lean. We believe the general direction of our works to be similar, but at the moment the typing judgement and the type checker are independent, thus providing no formal correctness or completeness guarantees. This aside, they make a novel choice of using single judgement for both typing and equality, while we pick a more conventional separate representations. Similar to Adjedj et al. [4] they implement MLTT with typed conversion and recursors.

Other related works fall in two camps. Stitch [27] and CakeML [49] are verified type checkers for simpler type systems, so they do not face many of the same challenges. Others formalise the specification and the metatheory - System DC [53] does this for Dependent Haskell, Abel et al. [2] focuses on decidability of conversion specifically, and Wieczorek and Biernacki [54] mechanise a normalisation-by-evaluation algorithm. These works are complementary to ours, they do meta-theory but do not develop a certified type checker, while we do the opposite.

7 Conclusion and Future Work

This paper presents a first step towards the goal of implementing a correct by construction type checker for a core language for Agda and moving from a trusted computing base to a trusted theory base.

In the process, we develop a set of techniques that can be useful for designing certified type checkers in general. We argue that using well-scoped syntax provides invaluable guidance, while not imposing too much of a proof burden on the developer. Using names rather than de Bruijn indices is useful for the same reason. Finally, we propose erasure annotations as an important tool to make the language developer more aware of the runtime behaviour of the code they are writing.

Future work There are many potential prospects to reach feature parity with Agda's internal language. Practically, we would like to add a pipeline to connect our existing development to Agda's compiler. We would also like to implement some core features of Agda, like indexed inductive datatypes, which would require a verified unification procedure, eta-equivalence, definitional irrelevance, and universe polymorphism. We plan to add termination and positivity checking, but we would like to procure certificates for these properties from Agda's compiler, which should simplify the implementation of the core checker.

Regarding applications, we would like to try using our embedded core language for type-safe metaprogramming [7, 25, 44] to automate tedious proofs, as well as for exchanging programs and proofs with other languages [18], enabling collaboration between different communities.

Acknowledgments. We would like to thank Lucas Escot for his contributions to the source code and valuable discussions. Jesper Cockx holds an NWO Veni grant on 'A trustworthy and extensible core language for Agda' (VI.Veni.202.216).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Bibliography

[1] Abel, A., Chapman, J.: Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types (Jun 2014). https://doi.org/10.4204/EPTCS.153.4

[2] Abel, A., Öhman, J., Vezzosi, A.: Decidability of conversion for type theory in type theory. Proceedings of the ACM on Programming Languages **2**(POPL), 23:1–23:29 (Dec 2017). https://doi.org/10.1145/3158111

[3] Adams, R.: Formalized Metatheory with Terms Represented by an Indexed Family of Types. In: Filliâtre, J.C., Paulin-Mohring, C., Werner, B. (eds.) Types for Proofs and Programs. pp. 1–16. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11617990_1

[4] Adjedj, A., Lennon-Bertrand, M., Maillard, K., Pédrot, P.M., Pujet, L.: Martin-Löf à la Coq. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 230–245. CPP 2024, Association for Computing Machinery, New York, NY, USA (Jan 2024). https://doi.org/10.1145/3636501.3636951

[5] Altenkirch, T., Kaposi, A.: Type theory in type theory using quotient inductive types. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 18–29. ACM, St. Petersburg FL USA (Jan 2016). https://doi.org/10.1145/2837614.2837638

[6] Ambros Kaposi: Towards quotient inductive-inductive-recursive types. In: 29th International Conference on Types for Proofs and Programs TYPES 2023 – Abstracts. pp. 124–126. Valencia (Spain), (2023-06-12/2023-06-15), https://types2023.webs.upv.es/TYPES2023.pdf#section.11.4

[7] Anand, A., Boulier, S., Cohen, C., Sozeau, M., Tabareau, N.: Towards Certified Meta-Programming with Typed Template-Coq. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving, vol. 10895, pp. 20–39. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94821-8_2

[8] Appel, A.W., Michael, N., Stump, A., Virga, R.: A Trustworthy Proof Checker. Journal of Automated Reasoning **31**(3), 231–260 (Nov 2003). https://doi.org/10.1023/B:JARS.0000021013.61329.58

[9] Atkey, R.: Syntax and Semantics of Quantitative Type Theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 56–65. LICS '18, Association for Computing Machinery, New York, NY, USA (Jul 2018). https://doi.org/10.1145/3209108.3209189

[10] Barendregt, H.: Introduction to generalized type systems. Journal of Functional Programming **1**(2), 125–154 (Apr 1991). https://doi.org/10.1017/S0956796800020025

[11] Bauer, A., Gilbert, G., Haselwarter, P.G., Pretnar, M., Stone, C.A.: Design and Implementation of the Andromeda Proof Assistant. In: DROPS-IDN/v2/Document/10.4230/LIPIcs.TYPES.2016.5. Schloss Dagstuhl –

Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.TYPES.2016.5

[12] Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. Journal of Functional Programming **9**(1), 77–91 (Jan 1999). https://doi.org/10.1017/S0956796899003366

[13] Bodin, M., Chargueraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised JavaScript specification. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 87–100. POPL '14, Association for Computing Machinery, New York, NY, USA (Jan 2014). https://doi.org/10.1145/2535838.2535876

[14] Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**(5), 552–593 (Sep 2013). https://doi.org/10.1017/S095679681300018X

[15] de Bruijn, N.G.: The mathematical language AUTOMATH, its usage, and some of its extensions. In: Laudet, M., Lacombe, D., Nolin, L., Schützenberger, M. (eds.) Symposium on Automatic Demonstration. pp. 29–61. Springer, Berlin, Heidelberg (1970). https://doi.org/10.1007/BFb0060623

[16] Bruno Barras: Coq en coq. Rapport de Recherche 3026, INRIA (Oct 1996)

[17] Carneiro, M.: Lean4Lean: Towards a formalized metatheory for the Lean theorem prover (Mar 2024). https://doi.org/10.48550/arXiv.2403.14064

[18] Cauderlier, R., Dubois, C.: FoCaLiZe and dedukti to the rescue for proof interoperability. In: Itp. pp. 131–147 (2017). https://doi.org/10.1007/978-3-319-66107-0_9

[19] Chapman, J.: Type Theory Should Eat Itself. Electronic Notes in Theoretical Computer Science **228**, 21–36 (Jan 2009). https://doi.org/10.1016/j.entcs.2008.12.114

[20] Coquand, T.: An algorithm for testing conversion in type theory. In: Logical Frameworks, pp. 255–279. Cambridge University Press, USA (Oct 1991)

[21] Coquand, T., Huet, G.: The calculus of constructions. Information and Computation **76**(2), 95–120 (Feb 1988). https://doi.org/10.1016/0890-5401(88)90005-3

[22] Curien, P.L.: An abstract framework for environment machines. Theoretical Computer Science **82**(2), 389–402 (May 1991). https://doi.org/10.1016/0304-3975(91)90230-Y

[23] Danielsson, N.A.: Operational semantics using the partiality monad. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. pp. 127–138. ICFP '12, Association for Computing Machinery, New York, NY, USA (Sep 2012). https://doi.org/10.1145/2364527.2364546

[24] Danielsson, N.A.: Logical properties of a modality for erasure (2019), https://www.cse.chalmers.se/~nad/publications/danielsson-erased.pdf

[25] Devriese, D., Piessens, F.: Typed syntactic meta-programming. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming - ICFP '13. p. 73. ACM Press, Boston, Massachusetts, USA (2013). https://doi.org/10.1145/2500365.2500575

[26] Dunfield, J., Krishnaswami, N.: Bidirectional Typing. ACM Computing Surveys **54**(5), 98:1–98:38 (May 2021). https://doi.org/10.1145/3450952

[27] Eisenberg, R.A.: Stitch: The sound type-indexed type checker (functional pearl). In: Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell. pp. 39–53. Haskell 2020, Association for Computing Machinery, New York, NY, USA (Aug 2020). https://doi.org/10.1145/3406088.3409015

[28] Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF, Lecture Notes in Computer Science, vol. 78. Springer, Berlin, Heidelberg (1979). https://doi.org/10.1007/3-540-09724-4

[29] Jesper Cockx: Operations On Syntax Should Not Inspect The Scope. In: Eduardo Hermo Reyes, Alicia Villanueva (eds.) TYPES 2023 – Abstracts. pp. 138–140. Valencia, Spain (2023), https://types2023.webs.upv.es/TYPES2023.pdf

[30] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 207–220. SOSP '09, Association for Computing Machinery, New York, NY, USA (Oct 2009). https://doi.org/10.1145/1629575.1629596

[31] Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 179–191. POPL '14, Association for Computing Machinery, New York, NY, USA (Jan 2014). https://doi.org/10.1145/2535838.2535841

[32] Lawrence C. Paulson: Isabelle: The next 700 theorem provers. In: P. Odifreddi (ed.) Logic and Computer Science, pp. 361–386. A.P.I.C. Studies in Data Processing, Academic Press (1990), https://www.cl.cam.ac.uk/~lp15/papers/Isabelle/chap700.pdf

[33] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (Jul 2009). https://doi.org/10.1145/1538788.1538814

[34] Malecha, G.: Extensible Proof Engineering in Intensional Type Theory. Ph.D. thesis, Harvard University, Graduate School of Arts & Sciences., Cambridge, Massachusetts (Nov 2014), http://nrs.harvard.edu/urn-3:HUL.InstRepos:17467172

[35] McBride, C.: Elimination with a Motive. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R., Pollack, R. (eds.) Types for Proofs and Programs. pp. 197–216. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45842-5_13

[36] McBride, C.: I Got Plenty o' Nuttin'. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 207–233. Lecture Notes in Computer Science, Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_12

[37] Nipkow, T., Wenzel, M., Paulson, L.C., Goos, G., Hartmanis, J., Van Leeuwen, J. (eds.): Isabelle/HOL, Lecture Notes in Computer Sci-

ence, vol. 2283. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
[38] Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. Ph.D. thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden (2007), https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf
[39] Pfenning, F., Schürmann, C.: System Description: Twelf — A Meta-Logical Framework for Deductive Systems, vol. 1632, pp. 202–206. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_14
[40] Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems **22**(1), 1–44 (Jan 2000). https://doi.org/10.1145/345099.345100
[41] Pollack, R.: How to believe a machine-checked proof. In: Sambin, G., Smith, J.M. (eds.) Twenty Five Years of Constructive Type Theory, p. 0. Oxford University Press (Oct 1998). https://doi.org/10.1093/oso/9780198501275.003.0013
[42] Schäfer, S., Tebbi, T., Smolka, G.: Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving. pp. 359–374. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_24
[43] Sestoft, P.: Deriving a lazy abstract machine. Journal of Functional Programming **7**(3), 231–264 (May 1997). https://doi.org/10.1017/S0956796897002712
[44] Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., Winterhalter, T.: The MetaCoq Project. Journal of Automated Reasoning **64**(5), 947–999 (Jun 2020). https://doi.org/10.1007/s10817-019-09540-0
[45] Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., Winterhalter, T.: Coq Coq correct! verification of type checking and erasure for Coq, in Coq. Proceedings of the ACM on Programming Languages **4**(POPL), 1–28 (Jan 2020). https://doi.org/10.1145/3371076
[46] Sozeau, M., Forster, Y., Lennon-Bertrand, M., Nielsen, J.B., Tabareau, N., Winterhalter, T.: Correct and Complete Type Checking and Certified Erasure for Coq, in Coq (Apr 2023), https://inria.hal.science/hal-04077552
[47] Stark, K., Schäfer, S., Kaiser, J.: Autosubst 2: Reasoning with multi-sorted de Bruijn terms and vector substitutions. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 166–180. CPP 2019, Association for Computing Machinery, New York, NY, USA (Jan 2019). https://doi.org/10.1145/3293880.3294101
[48] Strub, P.Y., Swamy, N., Fournet, C., Chen, J.: Self-certification: Bootstrapping certified typecheckers in F* with Coq. ACM SIGPLAN Notices **47**(1), 571–584 (Jan 2012). https://doi.org/10.1145/2103621.2103723
[49] Tan, Y.K., Owens, S., Kumar, R.: A verified type system for CakeML. In: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages. pp. 1–12. IFL '15, Association for

Computing Machinery, New York, NY, USA (Sep 2015). https://doi.org/10.1145/2897336.2897344

[50] The Agda Development Team: Agda 2.6.4 documentation (2023), https://agda.readthedocs.io/en/v2.6.4/

[51] The Coq Development Team: The coq reference manual – release 8.18.0 (2023), https://coq.inria.fr/doc/V8.18.0/refman

[52] The Idris Development Team: Documentation for the idris language – version 1.3.4 (2020), http://docs.idris-lang.org/en/v1.3.4/

[53] Weirich, S., Voizard, A., de Amorim, P.H.A., Eisenberg, R.A.: A specification for dependent types in Haskell. PACMPL **1**(ICFP) (2017). https://doi.org/10.1145/3110275

[54] Wieczorek, P., Biernacki, D.: A Coq formalization of normalization by evaluation for Martin-Löf type theory. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 266–279. CPP 2018, Association for Computing Machinery, New York, NY, USA (Jan 2018). https://doi.org/10.1145/3167091

[55] Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 357–368. PLDI '15, Association for Computing Machinery, New York, NY, USA (Jun 2015). https://doi.org/10.1145/2737924.2737958