ExEl: Building an Elaborator Using Extensible Constraints

Bohdan Liesnikov
Delft University of Technology
Delft, The Netherlands
B.Liesnikov@tudelft.nl

Jesper Cockx
Delft University of Technology
Delft, The Netherlands
J.G.H.Cockx@tudelft.nl

ABSTRACT

Proof assistants and dependently typed languages such as Coq, Agda, Lean, and Idris can be used to ascertain the correctness of software with mathematical precision. While much research has been done on their theoretical foundations, their actual implementations have been studied to a much lesser extent. As a result, features that are not considered part of the theoretical foundations - such as implicit arguments and type classes - have their own bespoke implementation for each language, making for code bases that are hard to understand and maintain. To address some of the common problems in the implementations of dependently typed languages, we present a modular architecture for implementing the transformation from user-friendly surface syntax into a small and well-behaved core language, also known as elaboration. Our architecture is made modular through the use of an open datatype of constraints and a plugin system for solvers that work on these constraints, which means that each new feature is contained in its own module. We showcase our design with a proof-of-concept elaborator for a language with dependent types, implicit arguments, higher-order unification, and instance arguments.

CCS CONCEPTS

• **Software and its engineering → Functional languages**; *Software design engineering*; • **Theory of computation** → *Type theory*; • **General and reference** → Design.

KEYWORDS

dependent types, elaborators, elaboration, open datatype, extensible datatype, type-checking

ACM Reference Format:
Bohdan Liesnikov and Jesper Cockx. 2023. ExEl: Building an Elaborator Using Extensible Constraints. In *The 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), August 29–31, 2023, Braga, Portugal*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652561.3652565

1 INTRODUCTION

Statically typed languages allow us to catch large classes of bugs at compile-time by checking the implementation against its type signature. When the types are provided by the user they can be viewed as a form of specification, constraining the behaviour of the programs. This comes with the benefit of more static guarantees but with an increased toll on the user to supply more precise information about the program. Many languages choose to infer types as a result, but another option is to use the information in the types to infer parts of the program. This idea was aptly worded by Conor McBride as "Write more types and fewer programs." [41; 31, chap. 2.1] Some examples of this include overloaded functions in Java, implicits in Scala, and type classes in Haskell.

In dependently typed languages like Agda [34, 50], Coq [51] or Idris [8] the types can be much more precise. This allows us to infer even larger parts of the program from the type. Examples include implicit arguments in Agda, implicit coercions in Coq, and tactic arguments in Idris. The inference of parts of the program must not be fully automatic but can also be interactive or partially automatic. Examples of interactive inference are holes in Agda and proof obligations in Coq, while canonical structures [29] in Coq and program-synthesis for holes in Haskell [24] are partially automatic.

Each of these different inference features has its own algoritms and extension points, which often evolved organically over time together with the language, and are often not well isolated from each other. For example, implicit arguments and instance search in Agda can interact in unexpected ways [3]. Sized types in Agda [2] also come with their own solver that often interacts poorly with the regular solver for implicit arguments. In Coq, canonical structures are notorious for producing unpredictable results yet they were not properly documented for 15 years [29]. Lean 4 aims to allow the users to develop new surface-level features [27] using elaboration monads [13], somewhat akin to elaborator reflection in Idris [11], but Lean 3 was built in a more conventional way [12]. All these bespoke algorithms and their interactions put a toll on the language developer to specify and implement them and on the user to understand them.

The part of the implementation of a dependently typed language that is responsible for type checking user-facing surface syntax and inferring the parts that have been left implicit is known as the *elaborator*. One common piece of infrastructure used by elaborators are metavariables, also known as "existential variables" [51, chap. 2.2.1], which represent as-of-yet unknown parts of the program. Together with metavariables also comes unification, i.e. the ability to constrain two terms to be equal. Metavariables and unification are heavily used throughout many elaborators for infering implicit arguments and for general type-checking, making them sensitive towards changes in unification algorithms. Because of the complexity unification, breaking changes are often discovered only when run against a large existing project on CI, such as the Standard and Cubical libraries for Agda or the `unimath` library for Coq.

To move towards a cleaner and more maintainable model for implementing elaborators, we propose a new architecture for an extensible elaborator for dependently typed languages. Practically, our architecture allows each feature to be contained within one module, as opposed to being spread around the codebase. The implementations of the individual features interact with the rest of the elaborator through an API for providing new kinds of constraints and new solvers for these new constraints as well as existing ones. This allows developers to put all solvers related to one feature in one place, thus fulfilling our goal of modularity. The design also separates the 'what' the solvers are doing from the 'when', making the separation between different phases of elaboration explicit. As a result, this allows the developer to reason more easily about exceptions and asynchronicity during elaboration and add new features in a more isolated fashion.

Contributions.

- We propose a new design blueprint for implementing an elaborator for a dependently typed language that is extensible with new constraints and new solvers. It supports type classes (Section 6.2), implicit arguments (Section 6.1), implicit coercions, and tactic arguments (Section 6.3).
- We propose a new view on metavariables as communication channels for the solvers, drawing parallels with asynchronous programming primitives (Section 5.3).
- We decompose the usual components of an elaborator, like the unifier in Agda, into a suite of solvers which can be extended and interleaved by user-provided plugins (Section 5).
- Following the blueprint, we present a prototype implementation of a dependently typed language available at github.com/liesnikov/extensible-elaborator[1].

2 UNIFICATION, CONSTRAINT-BASED ELABORATION AND DESIGN CHALLENGES

Constraints have been an integral part of compilers for strongly typed languages for a long time [35]. For example, the implementations of both Haskell [56] and Agda [34, chap. 3] use constraints extensively. In the former case, they are even reflected and can be manipulated by the user [36, chap. 6.10.3; 17]. This has proven to be a good design decision for GHC, as is reflected for example in a talk by Peyton Jones [38], as well as in a few published sources [39, 56].

[1]Also archived at doi.org/10.4121/e74fd14a-da79-4686-97e8-143ac5e0858d.v1.

In the land of dependently typed languages constraints are often used in a much less principled manner. Agda has a family of constraints that grew organically, currently counting 19 constructors.[2] Idris technically has constraints, with the only two constructors being equality constraints for two terms and for two sequences of terms.[3] The same holds for Lean.[4] These languages either use constraints in a restricted, single-use-case manner – namely, for unification – or in an ad-hoc manner.

In this section, we demonstrate why a more methodical approach to constraints results in more robust elaborators across the board. In particular, we go over three typical challenges that come up when building a compiler for a dependently typed language and the way they are usually solved: the complexity of managing a global state of metavariables for unification (Section 2.1), dealing with different kinds of implicit arguments and their solvers (2.2), and user-facing extension points to unification (Section 2.3).

2.1 Unification in the presence of meta-variables

As mentioned in the introduction, in the process of type-checking we use unification to compare terms. The unification algorithms are often one of the most complex parts of a type checker, which stems from the desire of compiler writers to implement the most powerful unifier while being limited by the fact that higher-order unification is undecidable in general. Some of this complexity is unavoidable, but we can manage it better by splitting up the unifier into smaller modular parts. In practice, this means that one does not have to fit together an always-growing unifier but can instead write different unification rules separately.

As a real-world example, Agda's unification algorithm for solving implicit arguments is spread between about a hundred functions and 2200 lines of code[5]. Each of the functions implements part of the "business logic" of the unifier, but a large part of their implementation is just there to deal with bookkeeping related to metavariables and constraints:

1. They throw and catch exceptions, driving the control flow of the unification.
2. They compute blockers that determine when a postponed constraint should be re-tried.
3. They have special cases for when either or both of the sides equation or its type are metavariables, or for when they are terms whose evaluation is blocked on some metavariables.

Concretely, this code uses functions like `noConstraints`, `dontAssignMetas`, `catchConstraint`, and `patternViolation`

[2]./src/full/Agda/TypeChecking/Monad/Base.hs#L1157-L1191. Here and henceforth we shorten the links in footnotes to paths in the repository, the source code can be found at github.com/agda/agda/blob/v2.6.4/.
[3]./src/Core/UnifyState.idr at github.com/idris-lang/Idris2/blob/e673d0
[4]./src/Lean/Meta/Match/Basic.lean#L161 at github.com/leanprover/lean4/blob/0a031f
[5]./src/full/Agda/TypeChecking/Conversion.hs

which rely on specific behaviour of the constraint solver. Other functions like `reduceB` and `abortIfBlocked` force the programmer to choose between letting the constraint system handle blockers or doing it manually. These things are known to be brittle and pose an increased mental overhead when making changes, with a corresponding risk of introducing new bugs.

The unifier is called from many places throughout the type-checker: when checking applications of variables and defined functions, type annotations on lambda expressions, forced 'dot' patterns, macro calls, and various primitive operations of Cubical Agda. At the same time, it is unintuitive and full of intricacies as indicated by multiple comments[6].

We would like the compiler-writer to be able to separate managing constraints and blockers from the actual logic of the comparison function. If we zoom in on the `compareAtom` function, the core logic can be expressed in about 20 lines[7] of simplified code (ignoring advanced features like sized types, cumulativity, polarity, and forcing). This is precisely what we would like the developer to write.

```
case (m, n) of
  (Lit l1, Lit l2) | l1 == l2 -> return ()
  (Var i es, Var i' es') | i == i' -> do
      a <- typeOfBV i
      compareElims [] [] a (var i) es es'
  (Con x ci xArgs, Con y _ yArgs) | x == y -> do
      t' <- conType x t
      compareElims t' (Con x ci []) xArgs yArgs
  ...
```

The functions described above are specific to Agda but in other major languages we can find similar problems with unifiers being large pieces of code that are hard to understand. The sizes of modules with unifiers are as follows: Idris has 1.5kloc[8] of unification code, Lean 1.8kloc[9], and Coq 1.8kloc[10]. For Haskell, which is not a dependently typed language yet, but does have a constraints system [38], this number is at 2kloc[11].

2.2 Type-checking function application in the presence of implicit arguments

During the type-checking of function application, there may be different kinds of implicit arguments to infer, for example, instance arguments, or tactic arguments. If we start from a simple case of type-checking an application of a function symbol to regular arguments, every next extension needs to be handled as a special case.

[6] ./src/full/Agda/TypeChecking/Conversion.hs##L484-L485, ./src/full/Agda/TypeChecking/Conversion.hs#L541-L549
[7] ./src/full/Agda/TypeChecking/Conversion.hs#L550-L599
[8] ./src/Core/Unify.idr
[9] ./src/Lean/Meta/ExprDefEq.lean
[10] ./pretyping/evarconv.mli at github.com/coq/coq/blob/b35c06
[11] ./compiler/GHC/Core/Unify.hs at gitlab.haskell.org/ghc/ghc/-/blob/b81cd709d

Take Agda – when checking an application of a function with implicit arguments[12] we already have to carry the information on how the arguments will be resolved (e.g. through instance search) and then create a specific kind of metavariable[13] [34, chap. 3] for each of those cases.

For handling `auto` variables, Idris 2 [52, chap. 13.1] has to essentially inline the search procedure through a chain of elaboration function calls (`checkApp` to `checkAppWith` to `checkAppWith'`) to `makeAutoImplicit`[14]. This can accommodate interfaces (or type classes), but one can imagine that if a different kind of implicit was added, like tactic arguments or canonical structures, we would have to inline the search again, requiring a non-trivial modification to the elaboration mechanism.

While the codebases above show that it is certainly possible to extend languages with new features if the language was not written with extensibility in mind, this can lead to rather ad hoc solutions.

2.3 Extending unification

While writing a unifier is hard enough as it is, at times the developers might want to give their users the ability to extend the unification procedure.

Canonical structures [29, 42] are one example as they are in the overlap between type classes and unification hints. Adding them to a language that does not support them requires an extension of the unification algorithm with a rule that says that projection from a canonical structure is injective [29, eq. 1].

One could also provide means to do so manually in a more general case, by allowing users to declare certain symbols as injective. This is one of the features requested by the Agda users [4].

Another example of user-extensible unification can be found in extensions with rules for associativity and commutativity, as described in the thesis by Holten [23]. It required changes to 2000 lines of code[15]. We would like to make changes such as this more feasible.

2.4 Summary of the issues

The examples above show that when building a dependently typed language the core might be perfectly elegant and simple, but the features that appear on top of it complicate the design. While the code in existing implementations often relies on constraints, the design at large does not put them at the centre of the picture and instead views them primarily as a gadget. For example, Agda's constraint solver[16] relies on the type-checker to call it at the point where it is needed and has to be carefully engineered to work with the rest of the codebase.

[12] ./src/full/Agda/TypeChecking/Implicit.hs#L96-L128
[13] ./src/full/Agda/TypeChecking/Implicit.hs#L130-L149
[14] ./src/TTImp/Elab/App.idr#L224-L241
[15] github.com/LHolten/agda-commassoc/tree/defenitional-commutativity (sic)
[16] src/full/Agda/TypeChecking/Constraints.hs#L247-L298

3 A BLUEPRINT FOR EXTENSIBLE ELABORATORS

Our idea for a new design is to shift focus more towards the constraints themselves:

1. We give an API for raising constraints that can be called by the type-checker, essentially creating an "ask" to be fulfilled by the solvers. This is similar to the idea of mapping object-language unification variables to host-language ones as done by Guidi et al. [20], the view of the "asks" as a general effect [7, chap. 4.4], or the communication between actors [5].
2. To make the language more modular, we make constraints an extensible data type and give an API to define new solvers with the ability to specify what kinds of constraints they can solve. Since we're working in Haskell we encode constraints in the style of Data types à la carte [47].

In the examples in this paper, we follow the bidirectional style of type-checking. In practice, however, the design decisions are agnostic of the underlying system, as long as it adheres to the principle of stating the requirements on terms in terms of raising a constraint and not by, say, pattern-matching on a concrete term representation.

From a birds-eye view, the architecture looks as depicted in Figure 1. Its main characteristics are the separation of plugins/unification into independent pieces and a clear boundary between solver dispatcher and the rest of the system.

Let us walk through each part of the architecture in more detail. The type-checking begins by initializing the state and traversing the syntax. The traversal raises the constraints, and for the moment, the constraints are simply stored. As soon as we finish the traversal of a block (one declaration in our case), the solver dispatcher is called. It traverses the set of constraints, and tries to solve each active constraint by making calls to different plugins. Each plugin, whether user-supplied (`Plugin A`) or provided by us (`unification`) consists of a handler and a solver. The handler determines if the plugin can potentially solve a constraint, if so, the dispatcher runs the corresponding solver.

All components have some read access to the state, including handlers which might for example verify that there are no extra constraints on the metavariable. For the write access: syntax traversal writes new metavariables to the state and elaborated definitions; the solver dispatcher writes updated meta-information; solvers write solutions to the metavariables and can raise new constraints.

For the moment we need to recompile the project to include new plugins. This is not necessity and a system that dynamically loads plugins is possible to implement in a way that is similar to GHC Plugins[17] or Accelerate[18] [32].

[17] mpickering.github.io/plugins.html
[18] github.com/tmcdonell/accelerate-llvm/blob/master/accelerate-llvm-native/src/Data/Array/Accelerate/LLVM/Native/Link/Runtime.hs

4 DEPENDENTLY TYPED CALCULUS AND BIDIRECTIONAL TYPING

Now that we have described our architecture, we will demonstrate its design through a practical implementation. First, in this section, we describe the core of the type system we implement. The following sections will then describe the constraints and their solvers. We take pi-forall [57] as a basis for the system and extend it with metavariables in the core syntax and implicit arguments in the surface syntax. However, for all other purposes, we leave the core rules intact and, therefore, the core calculus too.

4.1 Basic language and rules

This is a dependently typed calculus that includes Pi, Sigma and indexed inductive types.

Here is the internal syntax term data type, apart from service constructor omissions, like a placeholder `TRUSTME`.

```
data Term =
  -- type of types Type
    Type
  -- variables x
  | Var TName
  -- abstraction \x. a
  | Lam (Bind TName Term)
  -- application a b
  | App Term Arg
  -- function type (x : A) -> B
  | Pi Type (Bind TName Type)
  -- Sigma-type { x : A | B }
  | Sigma Term (Bind TName Term)
  | Prod Term Term
  | LetPair Term (Bind (TName, TName) Term)
  -- Equality type  a = b
  | TyEq Term Term
  | Refl
  | Subst Term Term
  | Contra Term
  -- inductive datatypes
  | TCon TCName [Arg] -- types (fully applied)
  | DCon DCName [Arg] -- terms (fully applied)
  | Case Term [Match]
    -- metavariables
  | MetaVar MetaClosure
```

Equality is not defined as a regular inductive type but is instead built-in. The user has access to the type and term constructor, but not the ability to pattern-match on it. Instead, the language provides a `subst` primitive of type `(A x) -> (x=y) -> A y` and `contra` that takes an equality of two different constructors of an inductive type and produces an element of any type.

On top of the above, the language includes indexed inductive datatypes and case-constructs for their elimination. Indexed inductive datatypes are encoded as parameterised datatypes with an equality argument constraining the index, also known as "Henry Ford" equality [10].

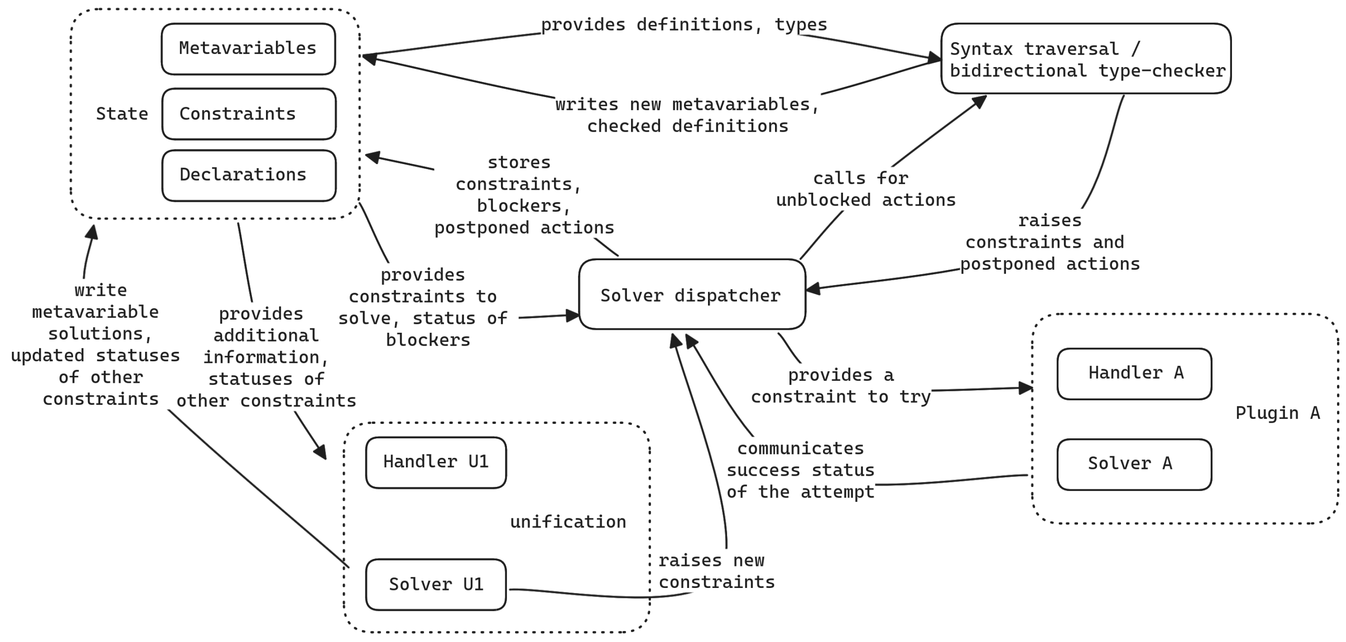
State
Metavariables
Constraints
Declarations
provides definitions, types
writes new metavariables, checked definitions
Syntax traversal / bidirectional type-checker
stores constraints, blockers, postponed actions
calls for unblocked actions
raises constraints and postponed actions
Solver dispatcher
write metavariable solutions, updated statuses of other constraints
provides additional information, statuses of other constraints
provides constraints to solve, status of blockers
provides a constraint to try
communicates success status of the attempt
raises new constraints
Handler U1
Solver U1
unification
Handler A
Solver A
Plugin A

Figure 1: Architecture diagram

As for metavariables `MetaVar`: as mentioned in the introduction, they are placeholders in the syntax tree (AST) that are produced in the process of elaboration. Metavariables do not appear in the surface syntax as they are not created by the user. In this paper we implement metavariables in the contextual style, as described by Abel and Pientka [1], therefore they are paired with a closure of type `MetaClosure`.

4.2 Syntax traversal

We implement the core of the elaborator as a bidirectional syntax traversal, raising a constraint every time we need to assert something about the type.

This includes the expected use of unification constraints, for example the case when we enter checking mode with a term whose type can be inferred:

```
checkType tm ty = do
  (etm, ty') <- inferType tm
  constrainEquality ty' ty I.Type
  return etm
```

It also includes any time we want to decompose the type provided in checking mode we pose a constraint that guarantees the type being in a specific shape:

```
checkType (S.Lam lam) ty = do
  mtyA <- createMetaTerm
  mtx <- createUnknownVar
  -- extend the context with a new variable signature
  mtyB <- extendCtx (I.TypeSig (I.Sig mtx mtyA))
                    (createMetaTerm)
  -- metaPi is the shape we want ty to be in
  let metaPi = I.Pi mtyA (bind mtx mtyB)
  constrainEquality ty metaPi I.Type
  -- rest of the traversal can now use mtyA and mbnd
  ...
```

At certain points, we have to raise a constraint which has an associated continuation. For example, when checking the type of a data constructor, the part of the program that comes as an argument to `constrainTConAndFreeze` will be suspended (or “blocked”) until the meta is solved with something of the shape `TCon _ _`.

```
checkType t@(S.DCon c args) ty = do
  elabpromise <- createMetaTerm
  constrainTConAndFreeze ty $ do
    mty <- substMetas ty
    case mty of
      (I.TCon tname params) -> do
        ...
      _ -> error "impossible"
```

We add one more case to the elaborator for implicit arguments of any kind, as will be described in more detail in Section 6.1.

```
checkType (Implicit) ty = do
  m <- createMetaTerm
  raiseConstraint $ FillInTheMeta m ty
  return m
```

As we see, the syntax traversal does not need to know anything about how implicit arguments are resolved. The only thing we require is that the elaboration of the argument is called with the type information available. This corresponds to how, in bidirectional typing, function application is done in the inference mode but the arguments are processed in checking mode.

```
inferType (App t1 t2) = do
  (et1, Pi tyA tyB) <- inferType t1
  et2 <- checkType t2 tyA
  return (App et1 et2, subst tyB et2)
```

5 CONSTRAINTS AND UNIFICATION

In Section 4 we described the syntax traversal part of the elaborator, which generates the constraints. In this section, we will go over the constraint datatype needed for the base language, how unification is implemented and how we can extend the unification procedure.

5.1 Base constraints

The datatype of constraints is open, which means the user can write a plugin to extend it. However, we offer a few constraints out of the box to be able to type-check the base language. For the base language, it suffices to have the following:

- A constraint that enforces equality of two terms of a given type
```
-- two terms given should be equal
data EqualityConstraint e =
     EqualityConstraint Term Term Type MetaVarId
```
Here, `MetaVarId` refers to the anti-unification variable (see Section 5.3).
- A constraint that ensures that a metavariable is resolved eventually:
```
-- this terms has to be filled in
data FillInTheTerm e =
     FillInTheTerm Term (Maybe Type)
```
- Lastly, a constraint which ensures that a term is a type constructor:
```
-- the term passed to the constraint
-- should be a type constructor
data TypeConstructorConstraint e =
     TypeConstructorConstraint Type
```

The type-checker raises constraints and supplies the necessary information in the process, but it is agnostic of how they will be solved. In all of the examples above, the parameter `e` is spurious. The need for this parameter comes from the technique we use to encode open datatypes, and other constraint types can use it to encode recursive occurrences of the constraints.

5.2 Interface of the solvers

On the solver side, we provide a suite[19] of solvers for unification that handle different cases of the problem. The most basic plugin is the `syntactic` plugin which resolves equality constraints where both sides are syntactically equal.

```
-- solves syntactically equal terms
syntacticHandler :: (EqualityConstraint :<: c)
                 => HandlerType c
syntacticHandler constr = do
  let eqcm = match @EqualityConstraint constr
  case eqcm of
    -- check for alpha-equality
    Just (EqualityConstraint t1 t2 ty _) ->
      return $ aeq t1 t2
    Nothing ->
      return False
syntacticSolver :: (EqualityConstraint :<: c)
                => SolverType Bool
syntactic :: Plugin
syntactic  = Plugin { solver  = syntacticSolver
                    , handler = syntacticHandler
                    ...
                    }
```

We first define the class of constraints that will be handled by the solver via providing a "handler" – a function that decides whether a given solver has to fire.[20] In this case, this amounts to checking that the constraint given is indeed an `EqualityConstraint` and that the two terms given to it are syntactically equal. Then we define the solver itself. In this case does not have to do anything except return `True` to indicate that the constraint is solved. This is because it shall only fire once it has been cleared to do so by the handler and the equality has already been checked. Finally, we register the solver by declaring it using a plugin interface.

The reason for the separation between a decision procedure and the execution of the solver is to ensure separation between a potentially slow, effectful solving and fast read-only decision-making in the handler. We opt for this division since handlers will be run on many constraints that do not fit them, therefore any write effects would have to be rolled back. Solvers, on the other hand, should only fire in cases when we can reasonably hope that the constraint will be solved and the effects will not have to be rolled back.

In a similar fashion, we define `leftMetaSolver` and `rightMetaSolver`, which only work on problems where one of the sides is a metavariable. Here the job of the solver is not as trivial – it has to check that the type of the other side indeed matches the needed one and then register the instantiation of the metavariable in the state.

Since a single constraint can often be handled by multiple solvers, we can provide priority preferences, using the `pre` and `suc` fields of the plugin interface. They are used to indicate whether the currently defined plugin should run before or after, respectively, which other plugins. At the time of running the compiler, these preferences are loaded into a big pre-order

[19] In the prototype we implement a subset of all unification rules, specifically: `identityPlugin`, `propagateMetasEqPlugin`, `reduceLeftPlugin`, `reduceRightPlugin`, `leftMetaPlugin`, `rightMetaPlugin`, `typeConstructorPlugin`, `typeConstructorWithMetasPlugin`, `piEqInjectivityPlugin`, `tyEqInjectivityPlugin`, `consInjectivityPlugin`, `typeInjectivityPlugin`, `unificationStartMarker`, and `unificationEndMarker`.

[20] The code shown above and in the rest of the paper is close to the actual implementation, but has been simplified for presentation purposes. `HandlerType a` and `SolverType a` both morally correspond to `(ConstraintF cs) -> SolverMonad Bool`.

relation for all the plugins, which is then linearised and used to guide the solving procedure.

```
rightMetaPlugin :: (EqualityConstraint :<: cs)
                => Plugin cs
rightMetaPlugin =
  Plugin { handler = rightMetaHandler
         , solver  = rightMetaSolver
         , symbol  = rightMetaSymbol
         , pre = []
         , suc = [leftMetaSymbol]
         }
```

5.3 Implementation of the solvers and unification details

We implement a system close to the one described by Abel and Pientka [1]. We modularise the implementation by mapping every function call in the simplification procedure to a raised constraint and every simplification rule to a separate solver. For example, the "decomposition of functions" [1, fig. 2] rule is translated to the following implementation.

```
piEqInjectivityHandler :: (EqualityConstraint :<: cs)
                       => HandlerType cs
piEqInjectivityHandler constr = do
  let eqcm = match @EqualityConstraint constr
  case eqcm of
    Just (EqualityConstraint pi1 pi2 _ _) ->
      case (pi1, pi2) of
        (I.Pi _ _ _, I.Pi _ _ _) -> return True
        _ -> return False
    _ -> return False
```

The handler is simply checking that both sides of the equality are indeed Pi-types, and in case either of the matches fails, it will be reported and the solver will not be fired. The `match` function above comes from the open datatype of constraints [47, sec. 5], checking if `constr` can be projected from `cs` to `EqualityConstraint`.

Now let us take a look at the solver. The rule we are implementing here states that two Π-types can only be equal if both the types of the domains (`a1`, `a2`) and the co-domains (`b1`, `b2`) are equal.

```
piEqInjectivitySolver :: (EqualityConstraint :<: cs)
                      => SolverType cs
piEqInjectivitySolver constr = do
  let (Just (EqualityConstraint (I.Pi a1 b1)
                                (I.Pi a2 b2) _ m)) =
        match @EqualityConstraint constr
  ma <- constrainEquality a1 a2 I.Type
  (x, tyB1) <- unbind b1
  (y, tyB2') <- unbind b2
  let tyB2 = subst y (I.Var x) tyB2'
      mat  = I.identityClosure ma
  mb <- extendCtx (I.TypeSig (I.Sig x e1 mat)) $
                 constrainEquality tyB1 tyB2 I.Type
  let mbt = bind x $ I.identityClosure mb
  solveMeta m (I.Pi mat mbt)
  return True
```

Following pi-forall, we use the unbound-generics[21] library to deal with the names and binders (`unbind`, `subst` functions). As it will fire after a handler returns `True`, we can assume the pattern-matches will not fail.

First, we constrain the equality of the domain of the Pi-type: `a1` and `a2`. The seemingly spurious metavariable `ma` returned from this call serves as an anti-unification [40] communication channel. Every time an equality constraint is created we return a metavariable that stands for the unified term. This metavariable is used for unification problems that are created in the extended contexts – in this case second argument of the Pi-type, but also when solving equalities concerning two data constructors. We do this to tackle the "spine problem" [55, sec. 1.4] – as we operate according to the "well-typed modulo constraints" principle, essentially providing a placeholder that is guaranteed to preserve well-typedness in the extended context.[22] Finally, `ma` has to be applied to a closure, which will keep track of the delayed substitution.

Then we can constrain the co-domains of Pi-types in an extended context. In case one of the solvers the constraints created in the extended context might need to know the exact shape of `ma`, we can block on the metavariable later, freezing the rest of the problem until it is instantiated.

As for the actual unification steps, we implement them in a similar fashion to the simplification procedure. For example, `leftMetaSolver` below handles the case where the left-hand side of an equality constraint is an unsolved meta:

```
leftMetaSolver :: (EqualityConstraint :<: cs)
               => SolverType cs
leftMetaSolver constr = do
  let (Just (EqualityConstraint t1 t2 _ m)) =
        match @EqualityConstraint constr
      (MetaVar (MetaVarClosure m1 c1)) = t1
  mt2 <- occursCheck m1 t2
  case mt2 of
    -- indicates a failure in occurs-check
    Left e -> return False
    -- indicates a passed occurs-check
    Right t2 ->
      case (invertClosureOn c1 (freeVarList t2)) of
        Just s -> do
          let st2 = substs s t2
          -- apply the subsitution
          solveMeta m1 st2
          -- instantiate the anti-unification variable
          solveMeta m st2
```

[21] hackage.haskell.org/package/unbound-generics-0.4.3
[22] Tog$^+$ [54, 55] focuses on extending the unification algorithm for the case where two sides of equality might not be of the same type, which is also a problem relevant for us. Their main argument against the usage of anti-unification in Agda provided there is that it is bug-prone. We think that in Agda the problems were stemming from the fact that anti-unification was implemented separately from unification, in which case it is indeed hard to keep the two in sync. There is no such duplication in our case since unification and anti-unification are one.

```
          return True
        Nothing -> return False
```

Once the occurs-check returns and if it was successful, we apply the inverted closure to the right-hand side of the equality.

By splitting up the rules into individual, simple solvers we can compartmentalise the complexity of the unifier, making sure that each rule is as decoupled from the others as possible. This does not deteriorate the properties of the system but does not help to enforce them either. We talk more about the challenge of proving correctness in Section 7.

5.4 Extending unification

Now that unification is implemented let us create a simple plugin that makes certain symbols declared by the user injective [4]. The actual implementation is relatively simple and is not dissimilar to the Pi-injectivity solver we showed above.

```
userInjectivitySolver :: (EqualityConstraint :<: cs)
                      => SolverType cs
userInjectivitySolver constr = do
  let (Just EqualityConstraint
              (I.App (I.Var f) a)
              (I.App (I.Var g) b) _ m)) =
        match @EqualityConstraint constr
  if f == g
  then do
    ifM (queryInjectiveDeclarations f)
        (do
           ms <- constrainEquality a b I.Type
           solveMeta m ms
           return True)
        (return False)
  else return False
```

where `queryInjectiveDeclarations` simply scans the available declarations for a marker that `f` has been declared injective.

The only big thing left is to make sure that this solver fires at the right time. This can only conflict with the “decomposition of neutrals” rule, so we indicate to the solver dispatcher that our plugin should run before it:

```
userInjectivityPlugin :: (EqualityConstraint :<: cs)
                      => Plugin cs
userInjectivityPlugin =
  Plugin { ...
         , solver = userInjectivitySolver
         , symbol = PluginId $ "userInjectivity"
         , pre = [ unifyNeutralsDecomposition
                 , unificationEndMarkerSymbol]
         , suc = [unificationStartMarkerSymbol]
         }
```

This modification does not alter the core of the language.

6 CASE STUDIES

Once we implement basic elaboration and unification we can extend the language. This is where we make use of the fact that the constraints datatype is open.

We saw before in Section 2.2 that conventional designs require separate handling of different kinds of implicit variables. To simplify the design we would like to uniformly dispatch a search for the solution, which would be handled by a fitting solver. We can achieve this by communicating the kind of the meta through its type in the second argument of `FillInTheMeta m ty`. The solvers then match on the shape of the type of the metavariable and handle it in a case-specific manner: instance-search for type classes, tactic execution for a tactic argument, or waiting for regular unification to solve the metavariable for regular implicit arguments.

In this section, we will describe the implementation details of regular implicit arguments (Section 6.1) and the implementation of type classes added on top of the implicit arguments (Section 6.2). And finally (Section 6.3) we sketch the implementation of coercive subtyping and tactic arguments.

6.1 Implicit arguments

As we mentioned, the rule for `Implicit` had to be added to the syntax traversal part of the elaborator. In fact, we require not one but two modifications that lie outside of the solvers-constraints part of the system. The first one is, indeed, the addition of a separate case in the syntax traversal, however contained. The second one lies in the purely syntactical part of the compiler: we need the pre-processor to insert the placeholder terms in the surface syntax.

In particular, we need to desugar declarations of functions in the following way. For any declaration of a function with some implicit arguments, we wrap the type of each implicit argument with an `Implicit`:

```
def f : {A : Type} -> {a : A} -> B a -> C
```

becomes

```
def f : (A : Implicit Type)
     -> (a : Implicit (deImp A))
     -> (b : B (deImp a)) -> C
```

Then, for each function call we insert the corresponding number of implicit arguments, for example transforming `f b1` to `f _ _ b1`.

For basic usage of implicit arguments this suffices – as soon as we have the placeholders in the surface syntax we create the constraints in the `Implicit` case of the syntax traversal. The only solver that is needed in this case is a trivial one that checks that the metavariable has been instantiated in the end. The actual solving in this case is done by the unifier itself, and the constraint simply serves as a way to guarantee that all implicits are eventually instantiated.

```
fillInImplicitSymbol :: PluginId

fillInImplicitHandler :: (FillInImplicit :<: cs)
                      => HandlerType cs
fillInImplicitHandler constr = do
```

```
  let ficm = match @FillInImplicit constr
  case ficm of
    Just (FillInImplicit term ty) -> do
      case term of
        MetaVar (MetaVarClosure mid _) ->
          isMetaSolved mid
        _ -> return False
    Nothing -> return False

fillInImplicitPlugin :: (FillInImplicit :<: cs)
                     => Plugin cs
fillInImplicitPlugin = Plugin {
    solver = fillInImplicitSolver,
  , handler = fillInImplicitHandler
  , symbol = fillInImplicitSymbol
  , suc = []
  , pre = []
  }
```

There is a design choice to be made in the implementation of `Implicit A` and `deImp`. One option is to turn them into a constructor and a projection of a record type, and the other is to make them computationally equivalent to `id`. In the former case, we need to manually unwrap them both in the pre-processor and during the constraint-solving, but since the head symbol is distinct we can guarantee that other solvers will not match on it, unless explicitly instructed to. In the latter case, one has to be cautious of the order in which the solvers are activated, particularly in the case of different search procedures, should they be implemented. However, in the example above it does not make a difference.

A limitation of this scheme is that we can only support functions with "obvious" implicit arguments – i.e. those that appear syntactically in the declaration of the function. This is due to the fact that insertion of metavariables happens before any type information is available. For this reason Haskell-like impredicativity [44] in type inference cannot be supported. If a more comprehensive support for implicit arguments is desired, our system could be extended with support for first-class implicits [25].

6.2 Type classes

Next, let us implement a plugin that adds support for type classes by means of instance arguments [14]. As in the case for implicit arguments in general, we rely again on a pre-processor to insert placeholder arguments of type `Instance a` for each instance argument of type `a`.

Let us start by going through an example of the elaboration process for a simple term. For type classes we need a few declarations, listed below in Agda-like syntax:

```
plus  :  {A : Type} -> {{PlusOperation A}}
     -> (a : A) -> (b : A) -> A

instance BoolPlus : PlusOperation Bool where
  plus = orb
```

And the exemplary term itself is:

```
m = plus True False
```

1. First, the pre-processor eliminates the implicits and type class arguments. We end up with the following declarations:

   ```
   plus : (impA : Implicit Type)
       -> Instance PlusOperation (deImp impA)
       -> (a : deImp impA) -> (b :  deImp impA)
       -> deImp impA

   instanceBoolPlus : InstanceT PlusOperation Bool
   instanceBoolPlus =
     InstanceC (TypeClassC (Plus orb))

   m = plus _ _ True False
   ```

   Turning the declaration `instance BoolPlus` into a usage of the constructor `InstanceC` is precisely the part we need the pre-processor to do.
2. Next, we go into the elaboration of `m`. The elaborator applies `inferType (App t1 t2)` rule four times and `checkType (Implicit) ty` twice on the two placeholders. The output of the elaborator is

   ```
   m = plus ?_1 ?_2 True False
   ```

   The state of the elaborator now contains four constraints:

   ```
   C1: FillInTheTerm ?_1 (Implicit Type)
   C2: FillInTheTerm ?_2 (InstanceT PlusOperation
                                        (deImp ?_1))
   C3: EqualityConstraint (deImp ?_1) Bool Type
   C4: EqualityConstraint (deImp ?_1) Bool Type
   ```

   The first two correspond to implicit arguments, while the latter two are unification problems rendered into constraints.
3. Now we step into the constraint-solving world. First, the unifier solves the latter two constraints, instantiating `?_1` to `Implicit Bool`. `C1` is then discarded as solved since `?_1` is already instantiated to `Implicit Bool`. Next, the type class resolution launches a search for the instance of type `Instance PlusOperation Bool`.
4. This is where the type class plugin can take over. It transforms `C2: FillInTheTerm ?_2 (InstanceT PlusOperation Bool)` to `C5: InstanceSearch PlusOperation Bool ?_2`. `C5` then gets matched by the search plugin for concrete instances, simply weeding through available declarations, looking for something of the shape `InstanceT PlusOperation Bool`. Such a declaration exists indeed and we can instantiate `?_2` to `instanceBoolPlus`.

Now let us take a look at the plugin for the constraint system. It is contained in a single file `./exel/src/Plugins/Typeclasses.hs`. In it, we define a new constraint type `InstanceSearch`, a solver that transforms constraints of the shape `FillInTheType ? (InstanceT _)` to the new constraint, and finally, a solver for instance search problems. The reason to transform the original constraint to the new type is to ensure that no other solver will make an attempt at this problem, therefore modulating interactions with other plugins.

Finally, the implementation of search for concrete instances is quite simple:

```
instanceConcreteSolver constr = do
  let (Just (InstanceSearch tcn ty m)) =
        match @InstanceSearch constr
  alldecls <- collectInstancesOf tcn <$> SA.getDecls
  sty <- SA.substAllMetas ty
  case Map.lookup sty alldecls of
    Just i -> do
      SA.solveMeta m (I.Var i)
      return True
    Nothing -> return False
```

We conjecture that implementation of canonical structures [29] would be relatively simple in such a system due to the openness of both unification procedure and instance search.

6.3 Tactic arguments and coercions

In a similar fashion to the transformation of the `FillInImplicit` constraints to `InstanceSearch`, we can implement plugins for tactic arguments and coercive subtyping.

The former would be quite similar to what we saw in the previous section 6.2, except we would have to resolve `FillInImplicit` to a `RunTactic` constraint (or run the tactic directly). The question of how to actually run the tactics is independent of the constraint machinery, noting that we allow for running arbitrary type-checking actions during solving of constraints.

Coercive subtyping is of a slightly different nature. First, it would be the most reliant on a pre-processor out of all of the examples described above, since coercions could potentially be inserted around each function argument, and potentially also around heads of applications and types of abstractions [48]. However, blindly inserting placeholders for coercions in every position leads to constraints that cannot be resolved locally. Consider the following example:

```
f : (A : Type) -> A -> A
arg : ArgTy

t : ExpTy
t = f _ arg
```

This declaration would desugar to

```
t = coerce _ (f _ (coerce _ arg))
```

which gives rise to two constraints, where `Coercion A B` stands for the evidence of `A` being a subtype of `B`:

```
C1 : Coercion ArgTy ?1
C3 : Coercion ?1 ExpTy
```

Clearly, we can not solve the first constraint in isolation and need to consider all coercion constraints related to a particular type at the same time. Hence this feature would be anti-modular in the sense that in order to side-step the conflict between inference and coercions the solver would need to look at the whole graph of subtyping constraints at the same time. While we can accommodate such solvers, due to solvers having write access to the state, it becomes much harder to modulate interactions between different plugins. It also comes with a potential performance penalty, which we describe further in Section 7.

7 LIMITATIONS

While our design offers a lot of flexibility, it does not solve every problem. In this section, we describe a few examples of extensions that do not quite fit in this framework as well as more general limitations.

7.1 Handling of meta-variables outside of definition sites

After we elaborate a definition, there can still be unsolved metavariables in it. This presents us with a design choice. The first option is to generalize the definition over the metavariables and instantiate them per-use site, essentially considering the metavariables as additional (implicit) arguments to the definition. The second option is to leave them up to be solved later, which might make elaboration less predictable since now the use sites can influence whether a particular definition type-checks. The third option is to freeze the metavariables by considering them to be postulates, and report them as an error at the end of type checking.

In particular, the second option allows us to incorporate more involved inference algorithms into the system. For example, if we were to implement an erasure inference algorithm as described by Tejiščák [49], we would have to create metavariable annotations (described as “evars” in the paper) that can be instantiated beyond the definition site. The downside is that the meaning of a definition can then change depending on where they are used, making the outcome of elaboration depend on the file as a whole rather than the definition itself.

7.2 The language is only as extensible, as the syntax traversal is

Extensibility via constraints allows for a flexible user-specified control flow as soon as we step into the constraints world. However, not all features can be supported purely by adding constraints and solvers. For example, consider the following simplified lambda-function type-checking function[23] from Agda:

```
checkLambda' cmp b xps typ body target = do
  TelV tel btyp <- telViewUpTo numbinds target
  if size tel < numbinds || numbinds /= 1
    then dontUseTargetType
    else useTargetType tel btyp
```

Here Agda steps away from the bidirectional discipline and infers a (lambda) function if the target type is not fully known. In order to add such a rule to our implementation, we would have to update syntax traversal to be more flexible in where it allows lambda expressions to appear. The solution, in this case, is effectively to replicate what Agda is doing

[23] ./src/full/Agda/TypeChecking/Rules/Term.hs#L430-L518

by implementing each type-checking rule in inference mode, essentially factoring out `dontUseTargetType` in Agda's code snippet above.

Likewise, implementing the commutativity and associativity unifier plugins from Holten [23] requires modifications in the core language, since we two terms that are equal during elaboration have to equal during core type-checking too.

7.3 Lack of backtracking

We do not implement any backtracking in the solver dispatcher as it is now. This means that every step taken is committing to a specific choice to how to solve the constraint, which can be a limitation in cases where one would like to have backtracking – for example, Agda's instance arguments search (with `--overlapping-instances` flag) [50, chap. 3.18], as well as type classes in Lean [43].

Backtracking in principle could be achieved by tracking changes to the state of the elaborator and the production graph for constraints, but such a system would be rather awkward. Alternatively, controlled backtracking can be implemented within one solver, removing the extension points, but allowing arbitrary control flow within the algorithm.

7.4 Reliance on a pre-processor

This work crucially relies on a pre-processor of some kind, be it macro expansion or some other way to extend the parser with custom desugaring rules. In particular, in order to implement n-ary implicit arguments correctly and easily we need the pre-processor to expand them to the right arity, similar to Matita [48, chap. 5] and others [25, 44].

7.5 Eager reduction and performance

As the pre-processing step inserts a fair number of wrappers and un-wrappers for all implicits and even more for coercions, we can expect a performance penalty due to larger terms and spurious computation steps. In particular, we expect this to be a major concern for coercive subtyping.

As one possible mitigation, we suggest a discipline where constraint solvers latch on to non-reduced types and terms in constraints. With this discipline, we can borrow a trick from the implementation of Coq, where the wrappers and unwrappers are identity functions and `coerce f t` computes to `f t`. This also means that constraints can/have to match on unreduced types in the e.g. `FillInTheTherm`.

In fact, we already use this trick to an extent for a different reason – since the calculus allows only fully applied type constructors, we have to wrap each type class constructor in a lambda-abstraction for it to appear as an argument to `TypeClassT typeClassName argType`. Or, concretely, we have to define and use `PlusOperation' = \A . PlusOperation A` in place of `PlusOperation` in the elaboration example in Section 6.2.

7.6 Proving correctness

As soon as we allow the users to implement their own solvers, there is little we can say about the correctness of the system as a whole without imposing proof obligations on the plugin writers. This is simply a consequence of the fact that we do not invent a new unification algorithm, but rather provide means of easier implementation for it. We conjecture that our system satisfies the solution and type preservation properties – Theorems 2 and 3 from Abel and Pientka [1], respectively – assuming that each solver on its own satisfies these properties. For termination the situation is similar – we conjecture that if every solver in the system "reduces the weight" of the unification problem, in terms of Theorem 1 by Abel and Pientka [1], we can guarantee termination of the solving process as a whole.

8 RELATED WORK

We are certainly not the first ones to try to tackle the extensibility of a language implementation. Dependently typed language implementations usually consist of at least four parts: parser, elaborator, core type-checker, and code generation backend. The code generation part is currently irrelevant to our interests, since for a language to be specified usually means for specification of the core, anything that happens after the core does not extend the language, but rather tries to preserve its semantics in some form. Therefore we're left with three parts: parser, elaborator, and core type-checker.

We see parser or syntax extensibility as a necessary part of an extensible language. This problem has been studied extensively in the past and has a multitude of existing solutions. Macros are one of them and are used heavily in various forms in many established languages [50, 51, 53] and can be powerful enough to build a whole language around [9].

Core extensibility, on the other hand, appears to be a problem with too many degrees of freedom. Andromeda [6, 7] made an attempt at extensible definitional equality but is quite far from a usable dependently typed language. Agda's philosophy allows developers to experiment with the core but also results in a larger amount of unexpected behaviours. In general, modifications of the core rules will result in fundamental changes in the type theory, which can break plenty of important properties like soundness or subject reduction.

This leaves us with the question of the extensibility of an elaborator. We will make a division here between syntax traversals, constraint solving and all other features. The syntax traversal part of the elaborator is relatively stable and commonly implemented following a (roughly) bidirectional discipline[15, 34, 48], so there seems little reason to make it extensible.

GHC has a plugin system that allows users to dynamically add custom constraint solvers, but the type of constraints itself is not extensible[24] [38, 39, 56].

Coq [51], being one of the most popular proof assistants, invested a lot effort into user-facing features: work on tactics like a new tactic engine [46] and tactic languages (Ltac2 [37], SSReflect [18], etc.), the introduction of a virtual machine for performance [19] and others. However, the implementation is quite hard to extend. One either has to modify the source

[24] gitlab.haskell.org/ghc/ghc/-/wikis/plugins/type-checker/notes

code, which is mostly limited to the core development team, as seen from the contributors graph, or one has to use Coq plugin system, which is rather challenging, and in the end, the complexity of it gave rise to TemplateCoq/MetaCoq [30, ; 45]. While MetaCoq did open the possibility for some plugins [16, 28, 33] to be written in a simpler way, their capabilities are still limited.

Agda has historically experimented a lot with different extensions to both the type system and the elaborator, even though the design does not accommodate these changes naturally. Instead, each of these extensions is spread throughout many different parts of the code base[25].

Lean introduced elaborator extensions [27, 53]. They allow the user to overload the commands, but if one defines a particular elaborator it becomes hard to interleave with others. In a way, this is an imperative view on extensibility.

Idris [8, 11] appeared as a programming language first and proof-assistant second and does not provide either a plugin or hook system at all, except for reflection. Idris also focuses on tactics as the main mechanism for elaboration.

Turnstile+ by Chang et al. [9] uses macros to elaborate surface syntax to a smaller core. Macros allow them to modularly implement individual features, however combining different features requires the user to re-define all macros from scratch. This is the same problem as the one we mentioned for Lean.

TypOS [5, 21] is perhaps the closest to our work, but there are two important differences. First, it is a domain-specific language for building type-checkers, while our design is language-agnostic, as long as the host language can model extensible datatypes in some capacity. Second, their approach settles features of the language as they are decided by the main developer and does not concern future changes and evolution. Finally, we try to stay close to the designs of existing dependently typed languages and offer flexibility in terms of choices, while TypOS requires the developer to start from scratch and restricts certain capabilities like overlapping rules for unification.

9 FUTURE WORK

We see three main prospects for future work:

- **Exploration of different kinds of metavariables.** Currently, we implement metavariables only for terms, while for some applications such as erasure [49] or irrelevance inference, it would be beneficial to have metavariables representing erasure and relevance annotations. Additionally, we can introduce metavariables for names and implement data constructor disambiguation more simply, which would remove the current need to block on the expected type of overloaded constructors.
- **Rendering more elements of the elaborator as constraints.** Currently, components such as the occurs checking and reduction are simple function calls. Including them in the constraint machinery would make the implementation more uniform and allow users to extend them.
- **Error messages** are not the focus of present work, but it would be interesting to see if we can incorporate ideas by Heeren et al. [22] into our system.
- **Potential optimisations**. Currently, our system has a lot of room for potential optimisations. The first step would be to allow handlers to pass some information to their respective solvers, which is conceptually an easy change but technically requires introduction of an existential type in the plugin. Additionally, some expandable per-plugin store for the solvers would be useful, for example, to avoid recompututation of type class instances on every invocation. Somewhat more ambitiously, one can imagine a caching system for constraints, avoiding the need for solving the same constraint more than once. In particular, caching of reduction seems like it would be beneficial since we currently do a lot of redundant computations. However, the memory usage of such a caching system might be prohibitive. Finally, we would also like to explore possibilities for concurrent solving, similar to the plans of Allais et al. [5] to use LVars for representing metavariables [26].

[25]some recent examples: github.com/agda/agda/pull/6385, github.com/agda/agda/pull/6354.

ACKNOWLEDGMENTS

Jesper Cockx holds an NWO Veni grant on ‘A trustworthy and extensible core language for Agda’ (VI.Veni.202.216).

REFERENCES

[1] Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications (Lecture Notes in Computer Science)*, Luke Ong (Ed.). Springer, Berlin, Heidelberg, 10–26. https://doi.org/10.1007/978-3-642-21691-6_5

[2] Andreas Abel and Théo Winterhalter. 2016. An Extension of Martin-Löf Type Theory with Sized Types. In *TYPES'16*. Novi Sad, Serbia, 2.

[3] Agda users. 2018. Performance Regression · Issue #3435 · Agda/Agda. https://github.com/agda/agda/issues/3435

[4] Agda users. 2023. “Injective for Unification” Pragma · Issue #6546 · Agda/Agda. https://github.com/agda/agda/issues/6546

[5] Guillaume Allais, Malin Altenmuller, Conor McBride, Georgi Nakov, Nordvall Forsberg, and Craig Roy. 2022. TypOS: An Operating System for Typechecking Actors. In *TYPES*. Nantes, France, 3. https://types22.inria.fr/files/2022/06/TYPES_2022_paper_31.pdf

[6] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone. 2018. Design and Implementation of the Andromeda Proof Assistant. (2018), 31 pages. https://doi.org/10.4230/LIPICS.TYPES.2016.5 arXiv:1802.06217

[7] Andrej Bauer, Philipp G. Haselwarter, and Anja Petković. 2020. Equality Checking for General Type Theories in Andromeda 2. In *Mathematical Software – ICMS 2020 (Lecture Notes in Computer Science)*, Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff (Eds.). Springer International Publishing, Cham, 253–259. https://doi.org/10.1007/978-3-030-52200-1_25

[8] Edwin Brady. 2013. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 5 (Sept. 2013), 552–593. https://doi.org/10.1017/S095679681300018X

[9] Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2019. Dependent Type Systems as Macros. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 3:1–3:29. https://doi.org/10.1145/3371071

[10] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1863543.1863547

[11] David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 284–297. https://doi.org/10.1145/2951913.2951932

[12] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[13] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 (Lecture Notes in Computer Science)*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[14] Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 143–155. https://doi.org/10.1145/2034773.2034796

[15] Francisco Ferreira and Brigitte Pientka. 2014. Bidirectional Elaboration of Dependently Typed Programs. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP '14)*. Association for Computing Machinery, New York, NY, USA, 161–174. https://doi.org/10.1145/2643135.2643153

[16] Yannick Forster and Fabian Kunze. 2019. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2019.17*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2019.17

[17] GHC development team. 2022. GHC 9.4.2 User's Guide. https://downloads.haskell.org/ghc/9.4.2/docs/users_guide/index.html

[18] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2008. *A Small Scale Reflection Extension for the Coq System*. Report. https://hal.inria.fr/inria-00258384

[19] Benjamin Grégoire and Xavier Leroy. 2002. A Compiled Implementation of Strong Reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 235–246. https://doi.org/10.1145/581478.581501

[20] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. 2017. Implementing Type Theory in Higher Order Constraint Logic Programming. https://hal.inria.fr/hal-01410567

[21] Guillaume Allais, Malin Altenmüller, Conor McBride, Georgi Nakov, Fredrik Nordvall Forsberg, and Craig Roy. 2022. TypOS. MSP group. https://github.com/msp-strath/TypOS

[22] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the Type Inference Process. *ACM SIGPLAN Notices* 38, 9 (Aug. 2003), 3–13. https://doi.org/10.1145/944746.944707

[23] Lucas Holten. 2023. *Dependent Type-Checking Modulo Associativity and Commutativity*. Master's thesis. Delft University of Technology, Delft, the Netherlands. https://repository.tudelft.nl/islandora/object/uuid%3A3f6c2de8-5437-4e47-b37f-6358c04eda9e

[24] James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 91:23–91:51. https://doi.org/10.1145/3547622

[25] András Kovács. 2020. Elaboration with First-Class Implicit Function Types. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 101:1–101:29. https://doi.org/10.1145/3408983

[26] Lindsey Kuper. 2015. *Lattice-Based Data Structures for Deterministic Parallel and Distributed Programming*. Ph. D. Dissertation. Indiana University, United States – Indiana. https://www.proquest.com/docview/1723378413/abstract/C592471EAAE94DBFPQ/1

[27] Leonardo de Moura and Sebastian Ullrich. 2021. Lean 4 - Metaprogramming. https://leanprover-community.github.io/lt2021/slides/leo-LT2021-meta.pdf

[28] Bohdan Liesnikov, Marcel Ullrich, and Yannick Forster. 2020. Generating Induction Principles and Subterm Relations for Inductive Types Using MetaCoq. https://doi.org/10.48550/arXiv.2006.15135 arXiv:2006.15135

[29] Assia Mahboubi and Enrico Tassi. 2013. Canonical Structures for the Working Coq User. In *Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP'13)*. Springer-Verlag, Berlin, Heidelberg, 19–34. https://doi.org/10.1007/978-3-642-39634-2_5

[30] Gregory Malecha. 2014. *Extensible Proof Engineering in Intensional Type Theory*. Ph. D. Dissertation. Harvard University, Graduate School of Arts & Sciences., Cambridge, Massachusetts. http://nrs.harvard.edu/urn-3:HUL.InstRepos:17467172

[31] Conor McBride. 2005. Epigram: Practical Programming with Dependent Types. In *Advanced Functional Programming*, Varmo Vene and Tarmo Uustalu (Eds.). Vol. 3622. Springer Berlin Heidelberg, Berlin, Heidelberg, 130–170. https://doi.org/10.1007/11546382_3

[32] Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. 2015. Type-Safe Runtime Code Generation: Accelerate to LLVM. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/2804302.2804313

[33] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. 2023. Formalising Decentralised Exchanges in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 290–302. https://doi.org/10.1145/3573105.3575685

[34] Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. Dissertation. Chalmers University of Technology and Göteborg University, Göteborg, Sweden. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

[35] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory and Practice of Object Systems* 5, 1 (1999), 35–55. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1⟨35::AID-TAPO4⟩3.0.CO;2-4

[36] Dominic Orchard and Tom Schrijvers. 2010. Haskell Type Constraints Unleashed. In *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10)*. Springer-Verlag, Berlin, Heidelberg, 56–71. https://doi.org/10.1007/978-3-642-12251-4_6

[37] Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. In *The Fifth International Workshop on Coq for Programming Languages /*. Cascais, Portugal, 3. https://popl19.sigplan.org/details/CoqPL-2019/8/Ltac2-Tactical-Warfare

[38] Simon Peyton Jones. 2019. Type Inference as Constraint Solving: How GHC's Type Inference Engine Actually Works. https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/

[39] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming* 17, 1 (Jan. 2007), 1–82. https://doi.org/10.1017/S0956796806006034

[40] F. Pfenning. 1991. Unification and Anti-Unification in the Calculus of Constructions. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. 74–85. https://doi.org/10.1109/LICS.1991.151632

[41] PTOOP. 2022. Type Inference Is a Thought Trap. Write More Types and Fewer Programs. https://twitter.com/PTOOP/status/1490496253276340227

[42] Amokrane Saibi. 1999. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories*. Ph. D. Dissertation. Université Pierre et Marie Curie - Paris VI. https://theses.hal.science/tel-00523810

[43] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. 2020. Tabled Typeclass Resolution. https://doi.org/10.48550/arXiv.2001.04301 arXiv:2001.04301 [cs]

[44] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 89:1–89:29. https://doi.org/10.1145/3408971

[45] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* 64, 5 (June 2020), 947–999. https://doi.org/10.1007/s10817-019-09540-0

[46] Arnaud Spiwack. 2011. *Verified Computing in Homological Algebra*. Ph. D. Dissertation. Ecole Polytechnique X. https://pastel.archives-ouvertes.fr/pastel-00605836

[47] Wouter Swierstra. 2008. Data Types à La Carte. *Journal of Functional Programming* 18, 4 (July 2008), 423–436. https://doi.org/10.1017/S0956796808006758

[48] Enrico Tassi, Claudio Sacerdoti Coen, Wilmer Ricciotti, and Andrea Asperti. 2012. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science* Volume 8, Issue 1 (March 2012). https://doi.org/10.2168/LMCS-8(1:18)2012

[49] Matúš Tejiščák. 2020. A Dependently Typed Calculus with Pattern Matching and Erasure Inference. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 91:1–91:29. https://doi.org/10.1145/3408973

[50] The Agda Team. 2023. Agda User Manual. https://agda.readthedocs.io/_/downloads/en/v2.6.4/pdf/

[51] The Coq Development Team. 2022. The Coq Proof Assistant, Version 8.15.0. Zenodo. https://doi.org/10.5281/zenodo.5846982

[52] The Idris Team. 2021. The Idris Tutorial. https://docs.idris-lang.org/_/downloads/en/v1.3.4/pdf/

[53] Sebastian Ullrich and Leonardo de Moura. 2020. Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages. In *Automated Reasoning (Lecture Notes in Computer Science)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 167–182. https://doi.org/10.1007/978-3-030-51054-1_10

[54] Víctor López Juan. 2020. Tog+. https://framagit.org/vlopez/togt

[55] Víctor López Juan. 2021. *Practical Heterogeneous Unification for Dependent Type Checking*. Ph. D. Dissertation. Chalmers University of Technology. https://research.chalmers.se/en/publication/527051

[56] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

[57] Stephanie Weirich. 2022. Implementing Dependent Types in Pi-Forall. (Aug. 2022). https://raw.githubusercontent.com/sweirich/pi-forall/2022/doc/oplss.pdf