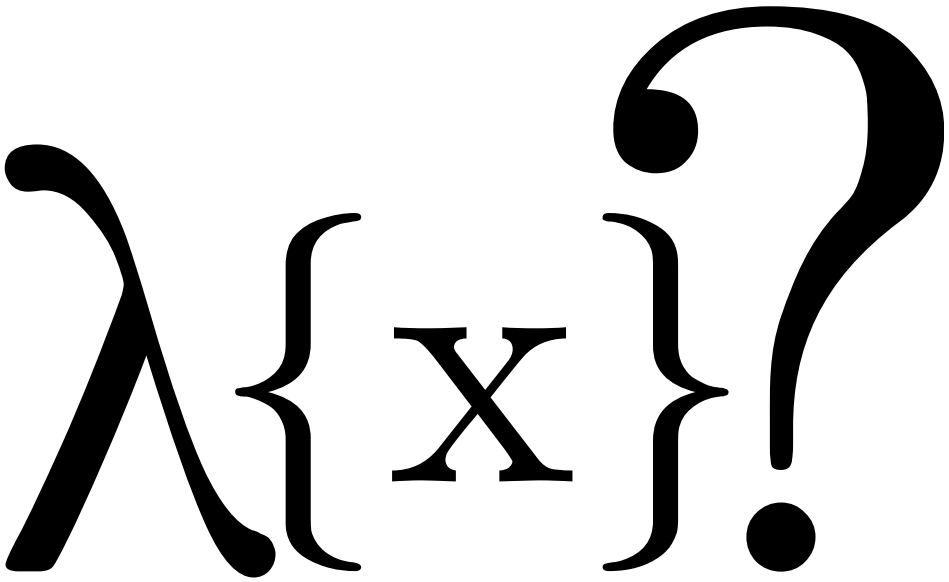
CHALMERS



Eliminating the problems of hidden-lambda insertion

Restricting implicit arguments for increased predictability of type checking in a functional programming language with dependent types

Master of Science thesis in Computer Science

MARCUS JOHANSSON
JESPER LLOYD

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2015

Eliminating the problems of hidden-lambda insertion

Restricting implicit arguments for increased predictability of type checking in a functional programming language with dependent types

MARCUS JOHANSSON
JESPER LLOYD

Examiner: Bengt Nordström

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

The image on the cover page is an abstract illustration of the fundamental problem of hidden-lambda insertion, as is outlined and treated in this thesis.

Department of Computer Science and Engineering
Göteborg, Sweden, March 2015

Abstract

Agda is a dependently typed functional programming language developed with the aim of making programming with dependent types practical. One highly useful feature of Agda, implicit arguments, allows a user to omit arguments which can be inferred by their relation to other arguments. This thesis aims to document a set of problems with the way implicit arguments are currently handled by Agda's type checker, and provide a way to resolve them. Three programs are presented as defining examples of the problems, along with derivations pinpointing the source of the problems within the checking process.

We conclude that although the problems cannot be resolved by modifying the heuristic central to this problematic type checking process, it is sufficient to impose a simple limitation on where implicit arguments may be declared, bound and given. The limitation somewhat restricts how things can be expressed, but we show that general expressiveness is not negatively affected.

A small dependently typed calculus called $\lambda_{\Xi\Phi}$ (Xiphi) is defined, and implemented in Haskell. The implementation supports the feasibility of the solution while also indicating shortcomings, which are discussed.

Keywords: Agda, dependent types, implicit arguments, type checking

Acknowledgments

We would like to thank our supervisor Andreas Abel for his patience in providing detailed explanations of the problems underlying this work, as well as his encouragement when we struggled to understand important concepts of the field. Andreas provided the ideas for the grouping-approach we investigated, and although we have done our best to distort his clearheaded ideas through constant experimenting and deliberation of our own, his original input is still very much recognizable in everything from the detailed problem analysis to the design of grammars and typing rules.

We would also like to thank Francesco Mazzoli, who helped us with the fundamentals of understanding constraint-gathering-based type checking of dependent types, and also provided much help and insight along the way.

Contents

1 Introduction

This section introduces a particular set of problems which can occur when type checking Agda programs that make use of implicit arguments. An extensive background section provides systematic descriptions of the concepts surrounding the topic, specifically relating to the usage and usefulness of implicit arguments, but it may be skipped by anyone who is already familiar with the subject. The background assumes some level of familiarity with basic concepts of functional programming. Additionally, the exact problems which are tackled in this work are outlined, followed by the approach taken to resolve them.

1.1 Background

Type systems serve multiple purposes. At the lowest level, successful type checking ensures some measure of run-time safety; well-typed programs are typically unable to execute illegal instructions. Types also enable programmers to abstract away from the details of implementation when constructing modular systems. In addition, type declarations often double as code-based documentation, making a program easier to read even in the absence of human-readable explanations [8].

The process of type checking is typically an integral part of the compilation scheme of a programming language, usually preceded by lexing and parsing phases, and succeeded by code generation or interpretation. In a functional programming setting, a basic task of a type checking algorithm is to decide whether or not a function definition adheres to its declaration, i.e., that its implementation matches the specification encoded in its type signature. To be of practical use, it is advisable that type checkers provide adequate error messages to guide the user in writing and debugging programs. It is also important that the type checking process terminates in a reasonable amount of time, ideally linear in relation to the program size, but never approaching anything exponential.

Rich type systems with elaborate type checking algorithms have evolved to accommodate the need for higher precision when encoding program specifications as types. Simple type systems may allow for little more than type annotations representing the memory layout of some data types, in which case the type checker is just an automated system for catching rudimentary bugs. More advanced type systems on the other hand may have a significant impact on how programs are written; features such as parametric polymorphism and automatic type inference allow types to be more expressive while relieving users from explicitly writing those type annotations that may be automatically derived. A good example of how these features are used in practice is the general-purpose functional programming language Haskell, which can infer the most general type of any expression[1] and lets the programmer write highly generic code with little effort. Integrating even more powerful type theoretical ideas into practically useful programming languages is an active area of research, of which this thesis seeks to be a part.

[1] When considering Haskell98 without extensions. Some modern extensions to the language may make the inference of some expressions undecidable.

1.1.1 Dependent types

While classical type systems allow for some precision when describing the domains and codomains of functions, they provide few means to encode details about structural transformations in a coherent and general way. Systems using dependent types allow for the definition of types which can be indexed by (depend on) terms in the language, providing a method for encoding properties of structures and functions directly through their type [1].

By using types to encode properties about term structures, the type signatures of functions can be used to encode properties about the transformations of those structures. This turns type checking of function definitions into correctness proofs for these definitions (for the properties defined in the signature). Although the user may have to aid the type checker when it comes to properties which cannot be automatically derived, this method of property declarations can allow for compiler-guided implementation by showing the user what types are required for pieces of code yet to be defined [6]. However, the inherent lack of general inference for dependent types, and the additional arguments for the indexing types, may require more effort on the part of the programmer.

```
-- Data type representing the natural numbers
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

-- Data type representing polymorphic vectors indexed by their length
data Vec (A : Set) : ℕ → Set where
  vnil  : Vec A zero
  vcons : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Figure 1 An Agda data type encoding the length of vectors at the type level

1.1.2 Agda: a dependently typed programming language

A prominent example of a language with full support for dependent types is Agda, which is a functional programming language developed with the aim of making programming with dependent types practical [7]. Agda is also a proof assistant with an interactive Emacs mode which facilitates the writing and checking of mathematical theorems and proofs expressed in the language.

These capabilities are in a sense just two sides of the same coin; Agda utilizes a close link between logic and computation known as the Curry-Howard correspondence, sometimes referred to as the propositions-as-types and proofs-as-programs interpretation. By this correspondence, the dependent types of Agda are naturally interpreted as quantification in intuitionistic logic. This allows for constructive mathematical propositions and proofs to be efficiently encoded in ordinary Agda types and programs, with type checking effectively corresponding to theorem proving.

A common example of dependent types employed in a more general-purpose manner is the types of vectors indexed by their lengths, presented in Figure 1. It shows how dependent types blur the boundary between term and type level

by allowing the vector type to depend on a value representing a natural number. This definition allows for turning indexing errors into type errors, meaning dependent types make it possible to statically guarantee the absence of out-of-bounds indexing. It should be noted that advanced type system features such as Haskell type classes have been used to simulate some aspects of dependent typing, results naturally lacking the brevity and clarity of languages supporting full dependent typing by design [5].

A couple of basic things relating to Agda's syntax and common practices are worth noting. In general the syntax resembles that of Haskell, a crucial difference being that the symbol `:` denotes a type declaration (instead of `::`). Local variables meant to represent terms are typically denoted by lower-case letters, while their declared type makes use of the corresponding capital letter; the binding `(a : A)` means that the argument has type `A`, and that the types of the succeeding arguments may depend on the value of this argument, by referring to its bind `a`. `Set` is short for $\texttt{Set}_0$ and refers to the type of simple types, the lowest level in an infinite hierarchy of type universes ($\texttt{Set}_0$ is contained in $\texttt{Set}_1$ which is contained in $\texttt{Set}_2$ etc.).

1.1.3 Implicit syntax

A major part of the practical usefulness of Agda lies within the ability to declare function parameters as implicit, which allows a user to omit the corresponding arguments when calling the function, instead letting the type checker try to infer the corresponding values (which may be terms or types). Such arguments may alternatively be provided explicitly by the user.

The archetypal use of implicit syntax in the realm of dependently typed languages is to omit type arguments in function applications. In the following paragraphs we will explore different ways of writing the polymorphic identity function, showcasing how implicit arguments may relieve the Agda user of unnecessary effort.

```
-- Explicit Agda id (could use _ instead of binding type to variable A)
id : (A : Set) → A → A
id A a = a
```

Figure 2 Agda identity function with an explicit type argument

Figure 2 shows a purely explicit version of the identity function written in Agda. A Haskell programmer might object to the seemingly unnecessary verbosity of this version of `id`; the type argument `A` is not even used in the RHS of the definition, but this function still demands that we provide the type argument. In Haskell, the symbol `_` (wild-card pattern) is used in cases when there is no need to bind an argument to a name, and the same syntax is available in Agda as well. However, ideally we would not want to mention the type argument at all, in neither arguments binds or function calls.

In Haskell all we need to tell the type checker is that the domain and codomain have the same type, as shown in Figure 3. This is made possible through a mechanism that generalizes type signatures by binding all type variables in universal quantifications scoped on the whole type. Haskell can safely do this since it by default assumes these type variables to be of kind *, whereas

```
-- Succinct Haskell id where polymorphism is fully implicit
id :: a → a
id x = x
```

Figure 3 Haskell identity function

in Agda we work with a stratified type universe and cannot make such an assumption. So how can we achieve a similar effect in Agda?

This is where implicit arguments come in. Agda allows for implicit arguments to be defined wherever regular arguments are allowed, by using curly brackets instead of parentheses. Making the type argument implicit we end up with the code shown in Figure 4. We are now allowed to omit the type from the pattern match as well as applications of the `id` function.

```
-- Implicit Agda id with type binding omitted
id : {A : Set} → A → A
id a = a
```

Figure 4 Agda identity function with an implicit type argument

1.1.4 Type checking implicit arguments using metavariables

Implicit syntax complicates the type checking process in two ways. First, the type checker now needs to determine where things are missing. Second, it needs to deduce what terms to put in the holes. Neither of these tasks is trivial; we introduce the basic mechanisms involved in this process by some examples.

```
-- Synonym for zero providing the type argument to id explicitly
zeroExpl : ℕ
zeroExpl = id {ℕ} zero
```

Simple type checking derivation of `zeroExpl`

1. Apply `id` to {ℕ}, instantiating `A` with ℕ in the resulting type:
   `id` {ℕ} : ℕ → ℕ
2. Checking if argument type matches. Trivial since we know that:
   `zero` : ℕ
3. Applying the function to `zero` thus gives correct type:
   `id` {ℕ} `zero` : ℕ

Figure 5 Type checking when giving an implicit argument explicitly

Consider the problem of type checking the functions `zeroExpl` from Figure 5, with `id` defined with an implicit type argument as seen in Figure 4. As shown by `zeroExpl`, Agda allows implicit arguments to be given explicitly with the use of curly brackets. The type checking of a function applied to explicitly given arguments is straightforward; apply arguments one by one, making sure that

the types of the arguments match what the function expects. The resulting type checking problem thus consists of little more than instantiating `A` from the type signature of `id` with $\mathbb{N}$ and concluding that `zero` is indeed of type $\mathbb{N}$.

When considering `zeroImpl` from Figure 6, we realise the type argument must be derived from information supplied somewhere else, in this case from the explicit argument `zero`. By inserting a kind of placeholder called a metavariable[2] in the empty space between `id` and `zero`, Agda is able to keep track of the missing piece, gathering information about how it is used. The names of metas are commonly preceded by an underscore, whereby the one considered in this problem is called `_N`.

```
-- Synonym for zero omitting the type argument to id
zeroImpl : ℕ
zeroImpl = id zero
```

Simple type checking derivation of zeroImpl

1. Insert a metavariable in place of the missing type argument.
   `zeroImpl = id {_N} zero`
2. Apply `id` to `{_N}`, instantiating `A` with `_N` in the resulting type:
   `id {_N} : _N → _N`
3. Checking if argument type matches requires that:
   `zero : _N`
4. A match would lead to the final type:
   `id {_N} zero : _N`
5. Solve metavariable `_N` through unification:
   `(zero : _N ∧ zero : ℕ) ⟹ _N = ℕ`
6. Instantiating the meta `_N` with $\mathbb{N}$ gives us the correct type:
   `id {ℕ} zero : ℕ`

Figure 6 Type checking when omitting an implicit argument

The process proceeds as with `zeroExpl`, applying `id` one argument at a time, until the only thing remaining is to decide a suitable value to substitute the metavariable `_N` for. This is done through a process called unification, which consists of solving a set of constraints collected from all the ways a metavariable has been used in the program under consideration. The solution to a unification problem is a substitution relating each metavariable to a term that can only depend on variables that were in scope when the metavariable was first introduced. Since the argument `zero` has the type $\mathbb{N}$, the constraint `_N = ℕ` is gathered. This constraint is easily solved by assigning $\mathbb{N}$ to `_N`, and thus the whole definition successfully type checks.

[2]The terms metavariable and *meta* will be used interchangeably, to keep descriptions succinct in certain places.

1.1.5 Tog type checking: separating elaboration and unification

Tog is an Agda-like language developed with the aim of creating a prototype language closely resembling Agda[3], but without the complexity brought on by the nature of Agda's development which has made it harder to understand and reason about. It constitutes a fully functional subset of Agda and was developed to provide insight into how the Agda type checker could be made simpler and more efficient.

Tog is of interest here since it implements a very compartmentalized idea of type checking where elaboration (the process of transforming expressions into terms while generating constraints) is a wholly separate stage from unification, unlike Agda where these processes are intertwined. This allows for very simple implementation of, and reasoning about, type checking and inference rules used in the elaboration stage.

1.2 Problem description

This work sets out to increase the *predictability* of type checking, but what is sought is not predictability in the strictest sense of the word; the procedures described have a deterministic set of rules. The notion of being predictable used here is that of being intuitively logical, transparent and consistent. Conversely, by *unpredictable* we mean behaviour which is unintuitive and inconsistent.

A number of issues affect the predictability of type checking in Agda; general type inference in polymorphic lambda calculus (System F) is undecidable [9, 2], as is general higher-order unification [3], two processes which are central to the core of Agda's type system and type checker respectively. Coupled with the problem of termination checking, these things make it difficult to be certain of the semantic correctness of a program, based on the result of checking alone. The type checking process is thus *incomplete* in the sense that programs which can be reasoned to be type correct will not pass type checking.

1.2.1 Avoidable unpredictability: hidden-lambda insertion

While general type inference and higher-order unification are inherently undecidable problems, the focus of this thesis is on a more manageable issue which nevertheless contributes negatively to the predictability of type checking in Agda (and also constitutes a source of incompleteness). This issue lies with the way implicit arguments are handled when type checking particular constructs, the end result being that some intuitively type correct definitions will fail to type check, and others that could be seen as incorrect will pass. The reason for this problematic behaviour is explained in some detail, starting with a simple overview of the relevant part of the type checking process and moving on to some concrete examples of programs which serve to illustrate different aspects of the problem.

The description of type checking implicit argument given in 1.1.4 is deliberately simplistic in that it only presents one part of the process when handling implicit arguments, namely the insertion of metavariables in the place of implicit arguments in an application. This manipulation essentially modifies the

[3]The code can be accessed at: https://github.com/bitonic/tog

expression in a way which allows the application to be structurally correct with regard to the expected argument (viewing the implicit argument as explicit).

Another type of term manipulation is required when coming from the other end of the equation, i.e. when implicit arguments need to be bound. This occurs in two places: function definitions and lambda abstractions. For a function definition, its declared type will guide whether or not binds for hidden (implicit) arguments need to be inserted. For every implicit argument without an explicit hidden binding[4], a bind will be inserted for that argument, up until the first unbound explicit argument (so called eager insertion). In the case of lambda abstractions, these will always have to be checked against a function type. When the domain of that function type is a hidden argument (a hidden function type), and the binder of the lambda abstraction is not hidden, a new binder for the hidden argument is inserted in order to allow the whole expression to match the actual type. We refer to this process as *hidden-lambda insertion*.

```
-- Standard boolean data type
data Bool : Set where
  true  : Bool
  false : Bool

-- Ty is a function type with type Set₁
Ty : Set₁
Ty = {T : Bool → Set}
     → ({b : Bool} → T b → T true) → T false

-- We use the axiom that some function f has the type Ty
postulate
  f : Ty

-- Simple synonym which type checks
works : Ty
works = f

-- Supposedly equivalent synonym which fails to type check
fails : Ty
fails g = f g

-- Manually inserted implicits makes the synonym type check again
hacks : Ty
hacks g = f (λ {b} → g {b})
```

Figure 7 Inconsistent behaviour when type checking implicit arguments

While Agda's heuristic way of handling implicit bindings necessitates some way of inserting hidden lambdas (just as it requires insertion of hidden arguments), it is also a source of unpredictability. Sometimes this process leads to undesirable results, as can be seen the example of Figure 7, which is modified from a real-world example given in a bug report [4]. One would intuitively ex-

[4] A hidden binding can be created by using the notation {v} to indicate that the variable v will be bound to an implicit argument corresponding to its order in the binds, or {n = v} to indicate that v will be bound to the implicit argument named n.

pect the synonym to type check (`fails = f` under an η-like reduction), but it requires an explicit insertion of a hidden binder and application.

There are also cases where the hidden-lambda insertion fails to produce the correct shape for an expression, when it is not immediately possible to determine whether the type of a scrutinized term should be regarded as having at least one implicit argument or none at all.

The decision problem of hidden-lambda insertion in Agda is made more difficult to analyze by the liberal placement rules for implicit arguments. It is a deliberate design decision to allow the placement of implicit arguments in the place of any explicit, which is justified as being necessary for reasons of expressiveness [7].

1.2.2 Three defining problems: Ty, Vec, Lam

We give three examples of programs in Agda, referred to as *Ty*, *Vec* and *Lam*, which present counter-intuitive behaviour of type checking. The first two cover scoping-related consequences of Agda's eager insertion of hidden lambdas, whereas *Lam* deals with premature inference of function types.

The first example, *Ty* (Figure 7), shows a case where a function synonym cannot be checked without manually inserting a hidden lambda and an implicit argument, despite the fact that the type of g can be shown quite easily to be compatible with f through direct comparison of their types. The expected and desired behaviour would naturally be to have the type checking handle this automatically, but the current strategy instead transforms g using hidden-lambda insertion in a way that leads to an unsolvable unification problem; the differently scoped structures created using this process lack a most general unifier.

```
-- The only difference between cons and vcons
-- is the order of arguments A and {n : ℕ}.
cons : {A : Set} → A →
       {n : ℕ} → Vec A n → Vec A (suc n)
cons a = vcons a
```

Figure 8 Vector constructor synonym with flipped arguments

The second example, *Vec* (Figure 8), on the other hand shows a case where a function synonym definition type checks despite the declared types being different, this too is a result of the eager insertion. While the *Ty* example demonstrates a case where an η-like expansion results in failed type checking, *Vec* shows how it may enable something which would otherwise not type check.

The third example, *Lam* (Figure 9), shows a case where type checking will fail as a result of handling non-linear type information. Looking at the type of `w`, it is easy to see that `b` should eventually be inferred to true, and the application of the type function `T` reduced to a type with an implicit argument. Furthermore, the lambda provided as argument to this function should be type correct assuming the insertion of a hidden lambda. However, due to the type checking procedure inferring the type of the lambda before knowing the result of `T b`, by the time the check can be made the types will have different shapes and fail to type check.

```
-- A simple definition of propositional equality
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a

-- Type function returning a function type
-- with or without an implicit argument.
T : Bool → Set1
T true  = {A : Set} → A → A
T false = Set → Set

-- The postulated functions we check against
postulate
  w : (b : Bool) → b ≡ true → T b → Bool
  f : (b : Bool) → T b → b ≡ true → Bool

-- A definition in which the lambda is checked
-- against T b after the equality constructor is checked
works : Bool
works = w _ refl (λx → x)

-- A definition in which the lambda is checked
-- before the equality constructor is checked
fails : Bool
fails = f _ (λx → x) refl
```

Figure 9 Order-dependent assumption of type shapes

The details of the derivations leading up to the problems described here can be found in A.1.2, A.1.3 and A.1.4 for *Ty*, *Vec* and *Lam* respectively.

1.3 Purpose

The purpose of this thesis is to devise a way of type checking implicit arguments which removes the unpredictability of hidden-lambda insertion and, having formulated such a method, reason about its use in a practical setting.

1.4 Method

In order to exemplify some of the problems related to type checking implicit arguments, three detailed type checking derivations are produced (three defining examples). An analysis of these derivations leads to the conclusion that the problematic behaviour cannot be avoided by simply changing isolated details of the type checking process.

For this reason, we define a dependently typed language, called $\lambda_{\Xi\Phi}$ (Xiphi), which places a restriction on the occurrence of implicit arguments, and show that this restriction leads to the desired behaviour for the defining examples while retaining sufficient expressiveness. In order to show this, we then define a calculus for type checking expressions in this new language. Finally, we encode and type check the three defining example programs in $\lambda_{\Xi\Phi}$ to see if they now behave as expected. To this end an implementation of the calculus in has been produced as part of the project.

1.5 Limitations

We do not consider certain practical details of language processing such as lexing and parsing, nor the process of providing error messages or similar feedback from the type checking process. Beyond that, there are four areas we consider out-of-scope for this project: unification, formal proofs, performance and the inclusion of stratified type universes.

An important mechanism used as part of the type checking, in the type systems we are working with, is the process of unification (a cursory description is given in Section 1.1.4). Unification is a large field which has undergone much study over the past decades, and while its use is central to the practical side of this project, the exact details of finding unifiers will not be considered. Instead we will mostly treat unification as a black box, considering the capabilities of existing algorithms as far as possible.

We do not provide any formal proofs of e.g. soundness or completeness of type checking in Xiphi. Instead, to indicate the correctness of our calculus when applied to the three defining problems, we provide detailed derivations in Appendix A.2.

Performance, although very important in practice for most aspects of language processing, has not been a point of focus when developing the solutions described in this thesis. The elaboration process described will be linear, which is most likely at the cost of an expensive unification process to complete type checking. Although the elaboration process may fail at the presence of certain identifiable structural errors in the input expressions, the general principle of failing as early as possible has not been given precedence over other design aspects. Some comments on this matter are made in the discussion, but no detailed analysis is provided.

In order to keep the solution (specifically the type checking rules) relatively simple, we do not handle stratified type universes in any way. While this does result in the system being logically inconsistent, this consistency is not crucial for the intended outcome of this project.

2 Design of the example calculus $\lambda_{\Xi\Phi}$ (Xiphi)

This section presents and motivates a way by which the problems described in the introduction may be approached[5], by defining a small dependent lambda calculus which will be the focus of the solution. It is explained how this calculus mitigates the problem of hidden-lambda insertion by enforcing a restriction on the occurrence of implicit arguments. The process of scope checking and type checking expressions in this language is then outlined; the details of the different phases are treated separately in Section 3 and 4.

2.1 Eliminating hidden-lambda insertion by uniformity

In order to overcome the unpredictable behaviour previously presented, we want to remove the possibility for the type checker to make any mistakes about whether a function type should or should not have implicit arguments. Agda's type checker has to make a decision based on immediately available data and cannot determine whether to insert hidden lambdas by the expression alone.

To remove the uncertainty of the decision, we can simply assume that implicit arguments will always be present, both declared and given, at particular points. By this a uniformity is achieved which allows the type checker to always state things with certainty based on the available structure. What is done is in fact to remove the decision altogether.

The idea is to always assume the presence of implicit structures in pairings: a telescope of implicit arguments before the domain of every function type, a sequence of implicit binders before every explicit binder in every lambda abstraction, and a sequence of given implicit arguments before every explicit argument in a function application. In summary, the explicit components of expressions will act as delimiters which indicate the occurrence of their implicit counterparts.

2.1.1 Encoding groups of implicits as expandable records

By using the fact that explicit constructs are delimiters we can safely package the preceding sequences of consecutive implicits into their own isolated language constructs. We choose to undertake this separation of the implicit structures in order to make the differences of their behaviour more obvious. The practical outcome of this choice is that we can define the behaviour of the implicit constructs separately from explicit ones, with the trade-off being that additional structures have to be defined internally to represent them.

Sequences of declared implicit arguments share many characteristics with dependent record types (which will also be referred to as *sigs*), and for this reason we choose to encode them as such. The only thing preventing us from stating that this representation is isomorphic to telescopes of implicits in Agda, is that record types have unique field names whereas the binders in an implicit telescope may shadow previous binders just as with ordinary explicit binders. To circumvent this we simply require all implicit binders to be unique within their group, allowing us to claim isomorphism between record types and sequences of implicit binds. We justify this restriction on the groups of implicits

[5]The approach used here was suggested by Andreas Abel, one of the main developers of Agda

in our language by the fact that things expressed using shadowing can always be expressed without it, and that having the same name within a group of implicits can be seen as confusing.[6]

The practical result of this encoding is that when the user declares a sequence of implicit arguments, these will be translated into a new explicit bind on a *sig*, with fields and types corresponding to the binds and types of the declaration (when there are no implicits, it will be the empty record type). Dependencies on the telescope will be translated into projections (for the corresponding fields) from that explicit bind.

In order for the isomorphism to hold we need record values (or simply *records*, also referred to as *structs*) to represent the values of the implicit arguments. However, since these values are generally not provided by the user in full (or even partially in most cases) it would be misleading to use the term record for the immediate result of the translation. Instead we call this structure an *expandable record* (which may be more conveniently read as *e-struct*), to indicate that it may not have all values available. Eventually, during a successful type checking, we expect these expandable records to be replaced with fully expanded records (containing values for all implicit arguments in the corresponding sig).

Additionally, we will use the related concept of an *expandable sig* (or *e-sig*) to define an expandable sequence of implicit binds preceding an explicit bind in a lambda abstraction.

2.2 Language processing: scope checking, type checking

We consider the entire process of verifying the correctness of an expression as consisting of two separate processes, which we call *scope checking* and *type checking*. The *scope checking* phase ensures that expressions are well scoped, but also performs the previously described encoding of implicit structures, the result of which is then the input of the *type checking* phase.

The type checking phase is itself a two-stage process. First, scope-checked and transformed expressions are processed further by a structurally recursive process, a stage which we, in the tradition of Agda, call *elaboration*. Elaboration of an expression results in two things: a term which approximates the original expression and may contain metavariables, and a set of constraints for the metavariables in that term. To complete the type checking process, the constraints have to be resolved and all metavariables have to be properly instantiated. This is the task undertaken during the *unification* stage.

2.2.1 Grammars: surface, core, internal

Scope checking and elaboration both perform transformations between different representations of expressions. Hence we have three separate grammars for the expressions constituting the input and output of these procedures: *surface*, *core* and *internal*.

The *surface* grammar (Figure 10) gives an indication of the user syntax, and is the input of scope checking. During scope checking, expressions in *surface* are transformed into corresponding expressions described by the *core* grammar (Figure 11), which makes use of the dependent record structures. Expressions

[6] For instance, it allows for expressions such as `f {a := x} {a := y} e`, where the `a`'s refer to different binders within the same group (the group associated with the domain of `f`).

in *core* are in turn the input of the elaboration stage, which outputs a term in the *internal* grammar (Figure 17), which is in turn the input of the unification stage.

Although the result of successful unification will technically be contained in a subset of *internal*, due to the absence of metavariables, we do not define a separate grammar for this subset. One can imagine this subset as the unstated grammar for finalized terms.

2.3 Grammar of the surface language

The main things achieved by the surface grammar is allowing function types (4), abstractions (5) and applications (6) to contain any number of implicit arguments per explicit argument, as indicated by the vector notation. The language may not be practically useful, but it is designed to be sufficiently expressive to encode the defining examples described in Section 1.2.2.

Constants (2) differ from variables (3) in that they are globally scoped, whereas variables are meant to only appear in some local context. The underscore (7) is used, as in Agda, when the user wants the value of an expression to be automatically inferred by the type checker. The category Φ_S is used to represent the two allowed ways of supplying an implicit argument explicitly; either by position (8) or by name (9). The identifiers k, a and b belong to a category capable of storing alphanumeric strings, the specific details of which are left unspecified.

Expressions

s, A, B, C	$::=$ Set	Type of types	(1)
	$\mid$ k	Constant	(2)
	$\mid$ a	Variable	(3)
	$\mid$ $\overrightarrow{\{a : A\}}\ (b : B) \to C$	Function type	(4)
	$\mid$ $\lambda\overrightarrow{\{a\}}\ b \to s$	Lambda abstraction	(5)
	$\mid$ $s_1\ \overrightarrow{\{\Phi_S\}}\ s_2$	Application	(6)
	$\mid$ _	Unknown expression	(7)

Field assignments

Φ_S	$::=$ s	Field assignment by position	(8)
	$\mid$ $a := s$	Field assignment by name	(9)

Identifiers

k, a, b	$::=$ $Ident$	Field and variable identifiers	(10)

Figure 10 Grammar of the surface language. The constructs surrounded by braces denote implicit declarations, bindings and arguments. The overset arrow indicates that vectors of these constructs are allowed, including empty ones.

3 Scope checking in $\lambda_{\Xi\Phi}$

This section starts off by describing the core grammar, which is the target of scope checking, and proceeds to provide the concrete rules for the scope checking procedure. Beyond ensuring well-scopedness of expressions, scope checking transforms the implicit structures of the surface language to record equivalents. Although we do not formally show semantic equivalence between input expressions and the result, the descriptions of the grammar and the procedure should be sufficiently detailed to indicate the correctness of the translation.

3.1 Grammar of the core language

The core grammar presented in Figure 11 is designed to handle all function types, lambdas and applications in a uniform way, whether they contain some implicit arguments or not. Expressions in core act as an intermediate representation used during elaboration, and are eventually transformed into *terms* in the *internal* language.

Expressions

e, D, E, F	$::=$ Set	Type of types	(11)
	$\mid$ k	Constant	(12)
	$\mid$ x	Variable	(13)
	$\mid$ $(x : D) \to E$	Function type	(14)
	$\mid$ $\lambda(x : D) \to e$	Lambda	(15)
	$\mid$ $e_1\ e_2$	Application	(16)
	$\mid$ $sig\{\overrightarrow{f : D}\}$	Record type	(17)
	$\mid$ $sig_e\{\overrightarrow{f}\}$	Expandable sig	(18)
	$\mid$ $struct_e\{\overrightarrow{\Phi_C}\}$	Expandable struct	(19)
	$\mid$ $e.f$	Projection	(20)
	$\mid$ _	Unknown expression	(21)

Field assignment

Φ_C	$::=$ e	Field assignment by position	(22)
	$\mid$ $f := e$	Field assignment by name	(23)

Identifiers

f	$::=$ $FieldId$	Field names	(24)
x, r	$::=$ $VarId$	Variable names	(25)
k	$::=$ $ConstId$	Constant names	(26)

Figure 11 Grammar of the core language. Successful scope checking transforms a surface expression into a well-scoped expression in the core language.

3.1.1 Differences from the surface language

The core grammar does not inherently provide the same restriction on the occurrence of implicit arguments as the surface grammar does. Function types now only present explicit binds, and applications lack the vector of provided implicit arguments, as these structures have now been provided with their own constructs: sig (17), e-sig (18) and e-struct (19). The addition of record types require projections (20) to create field references matching references to the original implicit bindings. Lambdas now have typed bindings, but as we will see, the only possible types in the binding are either an e-sig or an underscore.

One noticeable difference between the grammar of surface (Figure 10) and core is that the latter contains more constructs than the former, which may seem counter-intuitive. Although it *is* a reasonable argument against using the record structures at all, the benefits of being able to treat function types and applications in a more compact fashion is made apparent by the elaboration rules in Section 4.4.

It should also be noted that the restrictions from the surface grammar will carry over to expressions in core, when considering the handling of such expressions. Only a subset of core will be expressible through conversion from legal expressions in surface, meaning that expressions such as e.g $(x : Set) \to sig\{\}$ will never appear in core.

One important restriction not enforced explicitly by the grammar or the scope checking rules is that lambdas in application positions must be fully reduced prior to type checking. It is a reasonable restriction to not allow meaningless constructs in the grammar we are considering, hence any appearance of $(\lambda a \to e)\ b$ will not be allowed, and is assumed to have been reduced to $e[{}^{b}/_{a}]$ by some additional unspecified preprocessing.

3.2 Scope checking rules

The scope checking rules are defined in terms of a substitution context, an input structure and an output structure. Non-recursive requirements on the input structures, as well as certain non-recursive substructure transformations, are defined in side conditions for these rules.

In order to reduce the number of symbols, we overload the general structure of the rules, which will work between three pairs of different structure types. Rules are defined for transformations between entire expressions (from surface to core), as well as between telescopes of declared implicit arguments and vectors of given implicit arguments. The notation used can be found in Figure 12.

$$\Theta \vdash s \Rrightarrow e \qquad \text{General expression} \quad (27)$$

$$\Theta \vdash \overrightarrow{\{a : A\}} \Rrightarrow \overrightarrow{f : D} \qquad \text{Implicit argument telescope} \quad (28)$$

$$\Theta \vdash \{\Phi_S\} \Rrightarrow \Phi_C \qquad \text{Implicit argument vector} \quad (29)$$

Figure 12 Shape of the overloaded scope checking judgment

3.2.1 Substitution context

The context, Θ (Figure 13), is essentially a list of substitutions; variables in the surface expressions are replaced by corresponding expressions in the core language. Regular variable names will be used solely for references to explicit bindings, field names for references to implicit arguments within the same telescope, and projections for the same kind of references from outside the telescope.

$$
\begin{aligned}
\Theta ::= {} & \cdot && \text{Empty context} && (30)\\
\mid {} & \Theta, {}^{x}/_{a} && \text{Variable name (alpha convertible)} && (31)\\
\mid {} & \Theta, {}^{f}/_{a} && \text{Field name (not alpha convertible)} && (32)\\
\mid {} & \Theta, {}^{r \cdot f}/_{a} && \text{Projection} && (33)
\end{aligned}
$$

Figure 13 Grammar specifying the three different kinds of core constructs that variables from the surface language will be replaced with.

3.2.2 Variables

When encountering a variable (Figure 14), Θ will be traversed to look for the most recently added substitution for that variable. If no substitution is found, the expression is not properly scoped and scope checking will fail. Otherwise, the variable is replaced by the expression of the substitution.

$$
\textsc{Matching variable}\quad \frac{}{(\Theta, {}^{e}/_{a}) \vdash a \Rrightarrow e} \qquad (34)
$$

$$
\textsc{Non-matching variable}\quad \frac{\Theta \vdash a \Rrightarrow a'}{(\Theta, {}^{e}/_{b}) \vdash a \Rrightarrow a'}\ [a \neq b] \qquad (35)
$$

Figure 14 Scope checking rules for variables. A variable is only well-scoped when there is a matching substitution in Θ.

3.2.3 Type of types, constants and unknown expressions

Set, constants and underscores are not modified by the scope checking procedure and are always well-scoped, as shown in Figure 15. While obvious for *Set* and underscores, it could be considered an oversimplification to simply state that all constants are well-scoped. The practical consequences is that the existence of a constant implies that it is in scope where used, which is sufficient for the structures defined in our calculus. Constants carry the notion of being globally accessible.

$$
\textsc{Type of types}\quad \frac{}{\Theta \vdash \mathit{Set} \Rrightarrow \mathit{Set}} \qquad (36)
$$

$$
\textsc{Constants}\quad \frac{}{\Theta \vdash k \Rrightarrow k} \qquad (37)
$$

$$
\textsc{Unknown expression}\quad \frac{}{\Theta \vdash \_ \Rrightarrow \_} \qquad (38)
$$

Figure 15 *Set*, constants and underscores are always well-scoped

3.2.4 Function types, lambdas and applications

The rules for function types (39), abstractions (42) and applications (43) are defined in Figure 16. Function types are scope checked by first processing the telescope of implicits, the bindings contained therein being required to be locally unique. The result is placed in a dependent record type with a fresh binding, and added as the domain of the new core function type. The new codomain is produced by first scope checking the explicit bind under the context extended with projection substitutions for references to the implicit binds[7], and then doing the same for the codomain with the explicit bind also added to the context.

The types of the binds in implicit telescopes are scope checked in turn from first to last, with a substitution of the new field (of the bind) for the corresponding bind being added to context for each, before scope checking the remaining telescope. From this a scope checked vector of binds is produced.

As for lambda abstractions, a fresh binder is created for the vector of implicit binds (which, as for function types, have to be locally unique). The body of the lambda is then transformed under the context extended with projection substitutions for the implicit references and a fresh variable substituted for references to the explicit binding.

Since there are no direct dependencies between the constituent parts, the head, implicit arguments, and explicit argument, are scope checked individually, and the corresponding core application is rebuilt. The vector notation for the given implicit arguments indicate that they are all processed as once, with the resulting positional or named expressions being placed in an expandable struct.

An implicit argument may either be given by name or by position, but in either case, the given expression will be scope checked. If given by argument, the scope checked expression will be coupled with an assignment to a field corresponding to the original binding.

[7] The function *fieldify* is used to create fields corresponding to variable bindings.

FUNCTION TYPES WITH TELESCOPES OF IMPLICITS

$$\frac{\begin{array}{l}\Theta \vdash \overrightarrow{\{a : A\}} \Rrightarrow \overrightarrow{f : D} \\ (\Theta, \overrightarrow{r.f/a}) \vdash B \Rrightarrow E \\ (\Theta, \overrightarrow{r.f/a}, x/b) \vdash C \Rrightarrow F\end{array} \quad \left[\begin{array}{l}\text{fresh } r,\ x \\ \overrightarrow{a} \text{ distinct} \\ \overrightarrow{f} = \text{fieldify}(\overrightarrow{a})\end{array}\right]}{\Theta \vdash \overrightarrow{\{a : A\}}\ (b : B) \to C \Rrightarrow (r : sig\{\overrightarrow{f : D}\}) \to ((x : E) \to F)} \tag{39}$$

BASE CASE FOR IMPLICIT TELESCOPES

$$\frac{}{\Theta \vdash \epsilon \Rrightarrow \epsilon} \tag{40}$$

CONS FOR IMPLICIT TELESCOPES

$$\frac{\begin{array}{l}\Theta \vdash A \Rrightarrow D \\ (\Theta, f/a) \vdash \overrightarrow{\{a : A\}} \Rrightarrow \overrightarrow{f : D}\end{array}}{\Theta \vdash \{a : A\}, \overrightarrow{\{a : A\}} \Rrightarrow f : D, \overrightarrow{f : D}} \quad [f = \text{ fieldify}(a)] \tag{41}$$

LAMBDA ABSTRACTIONS

$$\frac{(\Theta, \overrightarrow{r.f/a}, x/b) \vdash s \Rrightarrow e \quad \left[\begin{array}{l}\text{fresh } r,\ x \\ \overrightarrow{a} \text{ distinct} \\ \overrightarrow{f} = \text{fieldify}(\overrightarrow{a})\end{array}\right]}{\Theta \vdash \lambda\overrightarrow{\{a\}}\ b \to s \Rrightarrow \lambda(r : sig_e\{\overrightarrow{f}\}) \to (\lambda(x : \_) \to e)} \tag{42}$$

FUNCTION APPLICATIONS

$$\frac{\Theta \vdash s_1, \overrightarrow{\{\Phi_S\}}, s_2 \Rrightarrow e_1, \overrightarrow{\Phi_C}, e_2}{\Theta \vdash s_1\ \overrightarrow{\{\Phi_S\}}\ s_2 \Rrightarrow (e_1\ struct_e\{\overrightarrow{\Phi_C}\})\ e_2} \tag{43}$$

IMPLICIT ARGUMENT GIVEN BY POSITION OR NAME

$$\frac{\Theta \vdash s \Rrightarrow e}{\Theta \vdash \{\Phi_S\} \Rrightarrow \Phi_C} \quad \left[\begin{array}{l}(\Phi_S, \Phi_C) \in \{(s, e),\ (a := s, f := e)\} \\ \text{where } f = \text{ fieldify}(a)\end{array}\right] \tag{44}$$

Figure 16 Scope checking rules for function types, abstractions and applications, which all make use of implicit substructure conversion.

4 Type checking in $\lambda_{\Xi\Phi}$

This section gives a thorough description of the type checking process, starting with an overview of the two comprising stages: elaboration and unification. The internal language, which is the target of elaboration and the domain of unification, is then described, followed by details of the contexts used and constraints generated in the course of type checking.

The property of well-typedness, a relation between terms and types in internal, is defined. Declarative rules which define the semantics of the entire type checking process are given, and related to subsequent definitions of algorithmic rules for it.

4.1 An overview of the type checking process

The type checking process is split up in two separate stages: elaboration and unification. In the same vein as Tog, we attempt to define these stages in isolation, although they are technically interleaved due to the precense of suspended type checking problems in the shape of checking constraints. This makes it easier to reason about properties of the individual stages, which is especially useful in this work, where the unification stage is not fully defined.

4.1.1 Elaboration

Elaboration is the process of transforming an expression in *core* to a well-typed term in *internal*, while producing a set of constraints for any metavariables in that term. It is performed under three different typing environments, which will be referred to as contexts: a metavariable- and constraint store (Ξ), a local variable context (Γ) and a fixed context for global constants (Σ). See Figure 18 for the notation used, and Section 4.2.2 for a detailed explanation.

At the top level, elaboration is initiated by checking an expression in *core* against a type in *internal*. This is done either by special checking rules which demand some specific goal type, or by inferring the type of the expression and deciding if the types are equal. If the equality relation can be determined immediately by alpha conversion and reflexivity, the expression is deemed well-typed and returned immediately. Sometimes however, equality cannot be determined straight away, whereby an equality constraint is generated.

Although there are type-level computations and reductions which are deferred to the unification stage, the elaboration does perform particular reductions to avoid the generation of unnecessary (trivially solvable) constraints.

4.1.2 Unification

The unification stage is concerned with solving the constraints gathered during elaboration. Traditionally, unification involves finding a most general unifier for unknowns in order to make two terms equal, the unknowns in this case being metavariables with fixed contexts. In this setting, we extend the notion of unification to include the handling of general checking constraints.

These checking constraints are to be seen as a paused type checking problems which may continue only when the shape of the type checked against has been revealed, most often through the instantiation of one or more metavariables.

As soon as the goal type is fully instantiated, the checking problem may be reconsidered using the ordinary checking rules available in elaboration.

The solving of equality constraints on the other hand requires some more attention to detail which is not central to the solution presented in this thesis. We provide declarative descriptions of equality and reductions which would guide this last step of unification in Appendix B.4, and in the following paragraphs we outline the idea of solving equality constraints, however we do not provide a full unification algorithm.

The basic idea is that the unifier should be able to compare two fully reduced terms structurally, component by component, and in the end either declare the terms equal or not. What makes this process interesting is the possible occurrence of metavariables, which the unifier needs to solve by instantiating them with adequate terms.

Valid instantations are found when a meta alone constitutes any side of an equality constraint. If the meta carries the empty substitution, it can be solved directly by instantiating it with the term on the other side of the equality constraint, by which Ξ must be updated accordingly. However if the meta carries a substitution, it may not be solvable straight away; it may be the case that it can only be solved by computing an inversion of the carried substitution, or by the resolution of some other constraint in Ξ.

Exactly how term equality is defined and decided, as well as how metas are instantiated and Ξ updated, is not very relevant to the problem at hand. Therefore we will assume some unspecified unification procedure capable of determining term equality to do the work for us, acting as an oracle. We assume the oracle can provide an answer for all unification problems thrown at it. While a considerable simplification, higher-order unifiers such as the one used by Agda should be able to replace this oracle in a real setting.

4.2 Internal language and type checking contexts

Understanding the grammar of the internal language is necessary to be able to read the declarations for well-typedness and understand the semantic descriptions of the type checking process. As the contexts are closely connected to the language, their descriptions are given here as well, along with a note on how substitutions and reductions are handled.

4.2.1 Grammar of the internal language

The grammar for terms in Figure 17 describes both output and input of the elaboration, the output being the transformed core expression and the input the type it was checked against.

While largely similar to surface and core, there are a number of important differences. Expandable sig types in the bindings of lambdas (49) have been replaced by full record types. The expandable struct of core has been replaced with a fully expanded struct (52), as all implicit arguments will be present for applications which expect them. Wildcards have been replaced by metavariables (54), which can carry stored substitutions (55) used to defer type computations. The substitutions are vectors of pairs providing a term t for every variable x.

Terms

t, u, v, T, U, V	$::=$ Set	Type of types	(45)
	$\mid$ k	Constant	(46)
	$\mid$ x	Variable	(47)
	$\mid$ $(x : U) \to V$	Function type	(48)
	$\mid$ $\lambda(x : U) \to v$	Lambda abstraction	(49)
	$\mid$ $t\ u$	Application	(50)
	$\mid$ $sig\{\overrightarrow{f : U}\}$	Record type	(51)
	$\mid$ $struct\{\overrightarrow{f := u}\}$	Record	(52)
	$\mid$ $t.f$	Projection	(53)
	$\mid$ $X[\overrightarrow{\sigma}]$	Metavariable	(54)

Substitutions

σ	$::=$ $t/_x$	Substitution	(55)

Identifiers

f, g	$::=$ $FieldId$	Field names	(56)
x, y, z, r	$::=$ $VarId$	Variable names	(57)
k	$::=$ $ConstId$	Constant names	(58)
X, Y	$::=$ $MetavarId$	Metavariable names	(59)

Figure 17 Grammar of the internal language. Successful type checking transforms a well-scoped core expression into a well-typed term in the internal language.

4.2.2 Contexts - variables, constants, metas and constraints

We use three contexts for the type checking phase, as seen in Figure 18.

The global context, Σ (60), contains a set of typed constants. In a complete system, defining larger program structures, it would contain the types of accessible external definitions, such as data types (type constructors), data constructors and functions, and are as such referred to as signatures.

Variable contexts, Γ, Δ (62), are used to store local variables along with their types. It is structurally identical to the global context, except for the binding being to variables rather than constants.

The type of a variable in Γ is allowed to depend on earlier variables, which means it is important that bindings are added in the right order. New variables are added to the rightmost side of Γ, to show the same order of dependence as in dependent function types.

Global context

$$\Sigma ::= \cdot \quad \text{Empty global context} \quad (60)$$
$$\mid \Sigma, k : T \quad \text{Signature} \quad (61)$$

Variable context

$$\Gamma, \Delta ::= \cdot \quad \text{Empty variable context} \quad (62)$$
$$\mid \Gamma, x : T \quad \text{Binding} \quad (63)$$

Metavariable context

$$\Xi ::= \cdot \quad \text{Empty metavariable context} \quad (64)$$
$$\mid \Xi, (X : T)[\Gamma] \quad \text{Metavariable introduction} \quad (65)$$
$$\mid \Xi, (u : U = T \dagger X)[\Gamma] \quad \text{Equality constraint} \quad (66)$$
$$\mid \Xi, (e \Leftarrow T \dagger X)[\Gamma] \quad \text{Checking constraint} \quad (67)$$

Figure 18 Grammar of global-, variable- and metavariable contexts

The metavariable context, Ξ (64), is used to store both metavariables and unification constraints. Over the course of type checking, when a new metavariable is needed a fresh ditto is added to Ξ with the current variable context. The context is used as input and output for all rules in the elaboration process to make it clear that it is being updated continuously. Since this behaviour is the same for all rules, one could alternatively omit it entirely except for when it needs to be updated or referenced.

Meta introductions (65) consist of a meta X of type T, with context Γ. All constraints also carry with them fixed variable contexts for a few reasons, mainly to maintain the well-typedness of the constituent terms and other constructs.

The equality constraint (66) describes a blocking wherein the meta X may only be instantiated by the term u when it has been determined that their types, T and U, are equal. This blocking is represented by the $\dagger$ symbol. Equality constraints are used when an expression is inferable, but the comparison between the inferred type and expected type cannot be decided directly.

The checking constraint (67) is similar to the equality constraint. The difference is that a checking constraint may block on any type checking problem which, due to the nature of the expression to be checked, needs a specific goal type to be able to continue.

4.2.3 A note on substitutions, reductions and termination

By computing substitutions on terms directly (and reducing the results), we are able to greatly reduce the number of metas and constraints generated in the elaboration process. The notation $t[\sigma]$ (or $t[{}^{u}/_{x}]$) used later on in e.g. well-typedness rules for applications, structs and projections, thus stands for the result of applying a substitution σ on the term t, with the result always conforming to the internal grammar. The details of this process is presented in Appendix B.3, but the main idea is that all free occurrences of the variable x in t are replaced with the term u, which by having globally unique identifiers for local variables amounts to all occurrences of x in t.

Computing a substitution may give rise to terms which are further reducible, either by introducing a lambda as the head of an application, or a struct as the target of a projection. Computing a projection is done simply by looking up and returning the value of the matching field in the record. This look-up will always succeed since the projection is only well-typed if the name of the projected field exists in the record type. Thus, in contrast to beta reduction of lambdas, any reducible projection produced as the result of a computed substitution is reduced straight away.

The rules for computing projections are presented in Appendix B.4, along with the standard rules for reducing and converting function, the usage of which is fully deferred to the unification stage. Although elaboration could also include reduction of lambdas, the practice of deferring it to unification means we are certain the elaboration procedure will always terminate, without having to further discuss the topic of termination checking. Thus the only possible source of undecidability is unification, the exact specification of which we will not consider in detail, but which should not have to be more powerful than the one currently employed by Agda.

4.3 Term-type relations and semantics of type checking

Two declarative relations are presented, a term-type relation for terms in the internal language, and an input-output relation for the type checking process as a whole. Together they provide an understanding of the output of type checking, along with the properties of that output.

$$\boxed{\Xi; \Gamma \vdash t : T}$$

Figure 19 Shape of the well-typedness judgment

4.3.1 Well-typedness of terms in the internal language

The well-typedness relation between terms and types in the internal language states that a particular term t should have a particular type T, under well-formed variable and meta contexts (see Figure 19). All term constructs have their own rules which are defined in terms of the same kind of contexts that are used in the elaboration rules.

For the contexts, a related predicate called well-formedness is defined, through well-typedness of their content. In short it entails that anything found in a variable context must be well-typed in terms of preceding items, and for the metavariable context, any term present in the constructs must be well-formed in terms of its local context. Exact rules can be found in Appendix B.1.

The rules for well-typedness are presented in three figures: basic constructs are handled in Figure 20, functions in Figure 21, and records in Figure 22. The rules for the constructs are grouped by a natural notion of relatedness, grouping basic constructs such as variables and constants, record-related constructs, function-related constructs etc. All well-typedness rules assume that contexts are well-formed and expressions well-scoped, and they entail the following three properties: T is a type (i.e. has type Set), t has type T, and the contexts remain well-formed.

TYPE OF TYPES

$$\frac{}{\Xi; \Gamma \vdash \mathit{Set} : \mathit{Set}} \tag{68}$$

CONSTANTS

$$\frac{}{\Xi; \Gamma \vdash k : T} \quad [(k : T) \in \Sigma] \tag{69}$$

VARIABLES

$$\frac{}{\Xi; \Gamma \vdash x : T} \quad [(x : T) \in \Gamma] \tag{70}$$

METAVARIABLES

$$\frac{}{\Xi; \Gamma \vdash X[\overrightarrow{\sigma}] : T[\overrightarrow{\sigma}]} \quad [(X : T)[\Delta] \in \Xi] \tag{71}$$

Figure 20 Well-typedness for basic constructs

1. **Type of types** (68): The type of *Set* is *Set*.

2. **Constants** (69): A constant k has the type specified by its binding in Σ given by the look-up relation $\Sigma(k)$, which of course presupposes that there exists a type binding for k in Σ.

3. **Variables** (70): Variables are treated the same way as constants, apart from the context being Γ instead of Σ.

4. **Metavariables** (71): The typing rule for metavariables assumes that $\overrightarrow{\sigma}$ is a vector of well-formed substitutions. It states that for a meta $X[\overrightarrow{\sigma}]$ to have the type $T[\overrightarrow{\sigma}]$ in Γ, the relation $X : T$ must hold under the context Δ of the substitution, which is the context within which the meta was originally introduced. Essentially what is stated is that well-typedness is preserved, under a new context, by valid substitutions.

5. **Function types** (72): Function types are of type Set, and are well-typed if their domain and codomain are well-typed, the latter under the context extended with a binding for the former.

6. **Lambda abstractions** (73): Lambda abstractions quite naturally have function types, the domain corresponding to the type annotation and the codomain to the type of the body, both of which must be well typed.

7. **Applications** (74): An application of t to u is well-typed if t has a function type and u is well-typed for its domain. Since the types are dependent, the resulting type of $t\ u$ is the codomain with a substitution for the binding of the domain, by the argument u.

8. **Record types** (75) (76): Well-typedness for record types is defined recursively in terms of deconstruction and reconstruction of sigs, covered in two cases. The base case consists of the empty record type which, like all types, have type Set. The recursive case states that a record type is well-typed if the type U of the first field f is well-typed, and the record type consisting of the remaining fields is well-typed under the context extended by the first field bound to its type. An unstated side condition here is that the field names f must be unique within the record type.

9. **Records** (77) (78): Records undergo similar deconstruction and reconstruction to record types. At this point it is important to remember that the records are fully expanded - containing values for all fields. The base case states that the empty record, unsurprisingly, has the empty record type. For the recursive case, in order for a record to be well-typed with a record type, the first field must correspond by name (by extension, all fields) and its term must be well typed with the type of the first field. The record constructed from the remaining fields must also have the type of the remaining record type, substituted with the value of the first field.

10. **Projections** (79): A projection is well-typed if the term that is the source of the projection (t) has a record type (i.e. is a record), and the projected field f is part of that type. The type of the field, U (look-up stated in the side condition), must carry the substitutions for all fields preceding the projected one, by the the values in those fields.

Function type

$$\frac{\Xi;\Gamma \vdash U : Set \quad \Xi;\Gamma, x : U \vdash V : Set}{\Xi;\Gamma \vdash (x : U) \to V : Set} \tag{72}$$

Lambda abstraction

$$\frac{\Xi;\Gamma \vdash U : Set \quad \Xi;\Gamma, x : U \vdash v : V}{\Xi;\Gamma \vdash \lambda(x : U) \to v : (x : U) \to V} \tag{73}$$

Application

$$\frac{\Xi;\Gamma \vdash t : (x : U) \to V \quad \Xi;\Gamma \vdash u : U}{\Xi;\Gamma \vdash t\ u : V[u/x]} \tag{74}$$

Figure 21 Well-typedness of function types, lambdas and applications

Empty record type

$$\frac{}{\Xi;\Gamma \vdash sig\{\} : Set} \tag{75}$$

Record type cons

$$\frac{\Xi;\Gamma \vdash U : Set \quad \Xi;\Gamma, f : U \vdash sig\{\overrightarrow{f : U}\} : Set}{\Xi;\Gamma \vdash sig\{f : U, \overrightarrow{f : U}\} : Set} \tag{76}$$

Empty record

$$\frac{}{\Xi;\Gamma \vdash struct\{\} : sig\{\}} \tag{77}$$

Record cons

$$\frac{\Xi;\Gamma \vdash u : U \quad \Xi;\Gamma \vdash struct\{\overrightarrow{f := u}\} : sig\{\overrightarrow{f : U}\}[u/f]}{\Xi;\Gamma \vdash struct\{f := u, \overrightarrow{f := u}\} : sig\{f : U, \overrightarrow{f : U}\}} \tag{78}$$

Projection

$$\frac{\Xi;\Gamma \vdash t : sig\{\overrightarrow{f : U}\}}{\Xi;\Gamma \vdash t.f : U[\overrightarrow{t.f/f}]} \quad [f : U \in \overrightarrow{f : U}] \tag{79}$$

Figure 22 Well-typedness of record types, records and projections

4.3.2 Declarative type checking rules

The declarative type checking rules define the behaviour of type checking by providing a description of the relation between the expressions, types and terms. Comparing this description to the algorithmic rules presented in 4.4 can best be done by considering the rules for inference and expression checking as being mappable to this relation, modulo a full unification procedure (this relation is defined for the complete type checking procedure. It should be noted that the algorithmic rules are defined in terms of input and output, whereas these relations are purely declarative, meaning that the thinking has to be adjusted when considering the different behaviour of inference and expression checking.

$$\boxed{\Xi; \Gamma \vdash e : T \rightsquigarrow t}$$

Figure 23 Judgment shape for the declarative type checking rules

The judgment used for the type checking semantics (Figure 23) should be read as: the expression e will be transformed to a term t with type T, under the contexts Ξ and Γ. Some of the rules make use of additional judgments which are defined near their point of use in the rules, with additional explanations provided. The rules for how expressions should be transformed into terms during type checking are presented in Figures 24 (basic constructs), 25 (functions), 26 (records) and 27 (struct expansion).

Type of types

$$\frac{}{\Xi; \Gamma \vdash \mathit{Set} : \mathit{Set} \rightsquigarrow \mathit{Set}} \tag{80}$$

Constants

$$\frac{}{\Xi; \Gamma \vdash k : T \rightsquigarrow k} \quad [(k : T) \in \Sigma] \tag{81}$$

Variables

$$\frac{}{\Xi; \Gamma \vdash x : T \rightsquigarrow x} \quad [(x : T) \in \Gamma] \tag{82}$$

Unknown expression

$$\frac{\Xi; \Gamma \vdash t : T}{\Xi; \Gamma \vdash \_ : T \rightsquigarrow t} \tag{83}$$

Figure 24 Transformation of basic constructs

1. **Set** (80)**, Constants** (81) and **Variables** (82): When type checked, these constructs are essentially unchanged, only being transformed to their internal counterparts. Their types are either fixed (in the case of *Set*) or determined by context-look-ups.

2. **Unknown expression** (83): Unknown expressions are transformed into well-typed terms (initially metavariables).

Function type

$$\frac{\Xi;\Gamma \vdash D : Set \rightsquigarrow U \quad \Xi;\Gamma, x : U \vdash E : Set \rightsquigarrow V}{\Xi;\Gamma \vdash (x : D) \to E : Set \rightsquigarrow (x : U) \to V} \tag{84}$$

Lambda abstraction

$$\frac{\Xi;\Gamma, x : U \vdash e : V \rightsquigarrow v}{\begin{array}{l}\Xi;\Gamma \vdash \lambda(x : D) \to e : (x : U) \to V \\ \rightsquigarrow \lambda(x : U) \to v\end{array}} \quad \left[\begin{array}{l}(D, U) \in \{(\_, U), \\ (sig_e\{\overrightarrow{f}\}, sig\{\overrightarrow{g : U}\})\} \\ \text{where } \overrightarrow{f} \subseteq \overrightarrow{g}\end{array}\right] \tag{85}$$

Application

$$\frac{\Xi;\Gamma \vdash e_1 : (x : U) \to V \rightsquigarrow t \quad \Xi;\Gamma \vdash e_2 : U \rightsquigarrow u}{\Xi;\Gamma \vdash e_1\ e_2 : V[u/x] \rightsquigarrow t\ u} \tag{86}$$

Figure 25 Transformation of functions

3. **Function types** (84): The term to which a function type expression will be transformed consist of the transformed domain and the transformed codomain (under the variable context extended by the domain binding).

4. **Lambda abstractions** (85): For function abstractions (lambdas) the rule is similar to the rule for function types, but the body does not have to be a type. The expandable sig is to be transformed into any full record type that contains the given fields as a subsequence.

   The side condition states that the types of the bindings can only be either underscores or expandable sigs. It also contains the additional condition that the declared implicit bindings must be a subsequence of the fields of the record type checked against.

5. **Applications** (86): For applications, the function- and argument expressions will be transformed into a term $t\ u$ in internal with type V. T must be a function type with U being its domain, the resulting term v simply being the application of the transformed expressions and its type the codomain of T with a substitution for its domain.

RECORD TYPE BASE

$$\frac{}{\Xi;\Gamma \vdash sig\{\} : Set \rightsquigarrow sig\{\}} \tag{87}$$

RECORD TYPE CONS

$$\frac{\Xi;\Gamma \vdash D : Set \rightsquigarrow U \quad \Xi;\Gamma, f : U \vdash sig\{\overrightarrow{f : D}\} : Set \rightsquigarrow sig\{\overrightarrow{f : U}\}}{\Xi;\Gamma \vdash sig\{f : D, \overrightarrow{f : D}\} : Set \rightsquigarrow sig\{f : U, \overrightarrow{f : U}\}} \tag{88}$$

EXPANDABLE STRUCT

$$\frac{\Xi;\Gamma \vdash \overrightarrow{\Phi_C} \rhd \overrightarrow{f : U} \rightsquigarrow \overrightarrow{f := u}}{\Xi;\Gamma \vdash struct_e\{\overrightarrow{\Phi_C}\} : sig\{\overrightarrow{f : U}\} \rightsquigarrow struct\{\overrightarrow{f := u}\}} \tag{89}$$

PROJECTION

$$\frac{\Xi;\Gamma \vdash e : sig\{\overrightarrow{f : U}\} \rightsquigarrow t}{\Xi;\Gamma \vdash e.f : U[\overrightarrow{t.f/f}] \rightsquigarrow t.f} \quad [f : U \in \overrightarrow{f : U}] \tag{90}$$

Figure 26 Transformation of records

6. **Record types** (87), (88): Empty record types are simply transformed to their internal counterpart. Non-empty record types will be transformed field-wise with progressively extended contexts.

7. **Expandable records** (89): Transformation of expandable records is a bit more involved than for the previous constructs. The type and transformed internal construct is determined by a special expansion judgment, which in turn depends upon a transformation of the vector of field assignments. This rule is defined in detail in Figure 27.

8. **Projections** (90): A projection $e.f$ is transformed by transforming e to a term t, which will have a record type containing a field f. As indicated by the side condition in the rule for projection-transformation, the type of the resulting projection $t.f$ will be the type for f in the type of t , with substitutions for all values (in the record), preceeding the projected field.

Struct expansion base

$$\overline{\Xi; \Gamma \vdash \epsilon \rhd \epsilon \rightsquigarrow \epsilon} \tag{91}$$

Struct expansion cons

$$\frac{\begin{array}{c}\Xi; \Gamma \vdash \overrightarrow{\Phi_C}(f : U) \rightsquigarrow (u, \overrightarrow{\Phi'_C}) \\ \Xi; \Gamma \vdash \overrightarrow{\Phi'_C} \rhd \overrightarrow{f : U[u/f]} \rightsquigarrow \overrightarrow{f := u}\end{array}}{\Xi; \Gamma \vdash \overrightarrow{\Phi_C} \rhd f : U, \overrightarrow{f : U} \rightsquigarrow f := u, \overrightarrow{f := u}} \tag{92}$$

Match

$$\frac{\Xi; \Gamma \vdash e : U \rightsquigarrow u}{\Xi; \Gamma \vdash (\Phi_C, \overrightarrow{\Phi_C})(f : U) \rightsquigarrow (u, \overrightarrow{\Phi_C})} \quad [\Phi_C \in \{e, f := e\}] \tag{93}$$

No match

$$\frac{\Xi; \Gamma \vdash \_ : U \rightsquigarrow u}{\Xi; \Gamma \vdash \overrightarrow{\Phi_C}(f : U) \rightsquigarrow (u, \overrightarrow{\Phi_C})} \tag{94}$$

Figure 27 Declarative struct expansion

9. **Struct expansion** (91) - (94):

   Struct expansion is defined recursively. The base case simply turns an empty e-struct into an empty struct.

   The recursive case behaves differently depending on whether the scrutinized field matches the field of the goal vector; either a field is specified (in which case that field will be retained) or there is only an expression to be typed and transformed. The premises state; the first field-type pair $(f : U)$ of the record will lead to a well-typed term, optionally consuming one item from the vector of typed field assignments; the remaining vector must be expanded to a record having the remaining record type with a substitution for the first value.

4.4 Algorithmic type checking rules

Here the operational side of the type checking process is defined in the form of algorithmic rules for elaboration of expressions. First the different kinds of elaboration judgments are explained, being grouped together in a way which isolates certain aspects, such as modification of the metavariable context. Then it is explained how the metavariable and constraint store Ξ is used during the elaboration stage, followed by checking and inference rules for all constructs in core are then presented and explained.

4.4.1 Elaboration judgment shapes: checking and inference

Here we present and explain the meaning of the main judgment shapes used for elaboration: checking and inference. The notation for these shapes can be seen in Figure 28. All types, terms and contexts generated by rules having these shapes should be well-typed and well-formed, given that the input is well-formed and well-scoped.

These algorithmic rules are similar in shape to the typing rules presented earlier, but instead of only describing a simple relation (type and term of an expression), they also provide notation and semantics for state updates. This is because the terms that are considered may be modified and new information may be added to the metavariable context (new metavariables and constraints). Constructs stated to the right of the '$\rightsquigarrow$' denote the elaborated term corresponding to the input expression, and a (usually) modified metavariable context.

The under-set plus (+) and minus (-) signs denote which components are part of the input to the judgment and which are output.

$$\underset{+}{\Xi};\underset{+}{\Gamma} \vdash \underset{+}{e} \leftleftarrows \underset{+}{T} \rightsquigarrow \underset{-}{t};\underset{-}{\Xi'} \qquad \text{Checking} \quad (95)$$

$$\underset{+}{\Xi};\underset{+}{\Gamma} \vdash \underset{+}{e} \rightrightarrows \underset{-}{T} \rightsquigarrow \underset{-}{t};\underset{-}{\Xi'} \qquad \text{Inference} \quad (96)$$

Figure 28 Judgments used for elaboration

For a checking judgment (95), the input required is a variable- and metavariable context as well as the expression to check and the type it is checked against. The result (the components to the right of the $\rightsquigarrow$) is a term t in the internal syntax corresponding to e, and an updated metavariable context. Type inference judgments (96) differ from checking judgments only in that the type is part of the output rather than the input.

4.4.2 Rules for generating metavariables and constraints

All extensions of the metavariable context in elaboration rules are done by using the rules presented in Figure 29 as premises. These rules make certain assumptions about their input, which when upheld by the caller entail well-formedness of the returned contexts.

An equality constraint is generated when the type of the expression to be elaborated may be inferred, but the resulting type equality between the inferred and checked type cannot be decided right away. It is preferable to decide any

decidable equalities as soon as possible, but we will not make any attempts at optimizing by failing this process early, with the consequence that some unnecessary equality constraints may be generated.

A checking constraint is generated when the shape of the goal type is unknown (e.g. if it is a metavariable), but the expression to be elaborated cannot be inferred because it relies on some special checking rule which makes direct use of the goal type. In that case we want to store away the checking problem until the goal type is known, and if it never becomes known even during unification, type checking will fail.

Introduce metavariable

$$\frac{}{\Xi; \Gamma \vdash \text{freshMeta}(T) \rightsquigarrow X; \Xi, (X : T)[\Gamma]} \quad [X \notin \Xi] \tag{97}$$

Generate equality constraint

$$\frac{\Xi; \Gamma \vdash \text{freshMeta}(T) \rightsquigarrow X}{\Xi; \Gamma \vdash \text{genEqC}(u : U = T) \rightsquigarrow X; \Xi, (u : U = T \dagger X)[\Gamma]} \tag{98}$$

Generate checking constraint

$$\frac{\Xi; \Gamma \vdash \text{freshMeta}(T) \rightsquigarrow X}{\Xi; \Gamma \vdash \text{genChkC}(e \leftleftarrows T) \rightsquigarrow X; \Xi, (e \leftleftarrows T \dagger X)[\Gamma]} \tag{99}$$

Figure 29 Operations for extending metavariable context

- **Introduce fresh metavariable with known type** (97): The function takes a type T (which is required to be well-typed, i.e., having type *Set*) and adds a fresh metavariable X to Ξ, returning that variable along with the updated context. By requiring T to be well-typed, and creating X fresh, the operation retains well-formedness of Ξ.
- **Generate equality constraint** (98): A type equality problem $u : U = T$ may be added to Ξ only if it has been made sure that u is of type U, (which implies that it U has type *Set*,) and if T is of type *Set*. A metavariable X with type T is returned to be used in place of u until the type equality $U = T$ has been decided, after which X may be instantiated with u.
- **Generate checking constraint** (99): This rule is very similar to the one for generating equality constraints, the difference being the problem blocked on: a checking problem $e \leftleftarrows T$, where T must have type *Set*. No demands are placed on the structure of e, since it is not yet elaborated, but in practice it will only be either a lambda with an binding typed by an expandable sig, or an expandable struct.

4.4.3 Rules for checking an expression against a type

Checking expressions in the core language (referring to the elaboration operation, and not the overall process) is done by using the rules in Figure 30, (with the expansion of structs detailed in Figure 31). If no special checking rule matches, the rule for checking a general expression found in Figure 32 is used, which relies on type inference and equality.

Check unknown expression

$$\frac{\Xi; \Gamma \vdash \text{freshMeta}(T) \rightsquigarrow X; \Xi'}{\Xi; \Gamma \vdash \_ \leftleftarrows T \rightsquigarrow X; \Xi'} \tag{100}$$

Check lambda

$$\frac{\Xi; \Gamma, x_1 : U \vdash e \leftleftarrows V[x_1/x_2] \rightsquigarrow v; \Xi'}{\Xi; \Gamma \vdash \lambda(x_1 : D) \to e \leftleftarrows (x_2 : U) \to V \rightsquigarrow \lambda(x_1 : U) \to v; \Xi'} \quad \left[\begin{array}{l} (D, U) \in \{(\_, U), \\ (sig_e\{\overrightarrow{f}\}, sig\{\overrightarrow{g : U}\})\} \\ \text{where } \overrightarrow{f} \subseteq \overrightarrow{g} \end{array} \right] \tag{101}$$

Generate lambda checking constraint

$$\frac{\Xi; \Gamma \vdash \text{genChkC}(\lambda(x : D) \to e \leftleftarrows T) \rightsquigarrow X; \Xi'}{\Xi; \Gamma \vdash \lambda(x : D) \to e \leftleftarrows T \rightsquigarrow X; \Xi'} \tag{102}$$

Check e-struct

$$\frac{\Xi; \Gamma \vdash \overrightarrow{\Phi_C} \rhd \overrightarrow{f : U} \rightsquigarrow \overrightarrow{f := u}; \Xi'}{\Xi; \Gamma \vdash struct_e\{\overrightarrow{\Phi_C}\} \leftleftarrows sig\{\overrightarrow{f : U}\} \rightsquigarrow struct\{\overrightarrow{f := u}\}; \Xi'} \tag{103}$$

Generate e-struct checking constraint

$$\frac{\Xi; \Gamma \vdash \text{genChkC}(struct_e\{\overrightarrow{\Phi_C}\} \leftleftarrows T) \rightsquigarrow X; \Xi'}{\Xi; \Gamma \vdash struct_e\{\overrightarrow{\Phi_C}\} \leftleftarrows T \rightsquigarrow X; \Xi'} \tag{104}$$

Figure 30 Special checking rules

1. **Unknown expressions** (100): When checking an unknown expression, a fresh metavariable of the correct type is introduced and returned.
2. **Lambdas** (101) (102): Lambdas are checked by first matching the types of the bindings; an expandable sig must be a subsequence of a record type while an underscore allows any type. The binding is added to context and the body is checked against the codomain of the goal type. If the shape of the goal type is not known, a checking constraint will be generated and a metavariable returned in place of an elaborated lambda.
3. **Expandable structs** (103) (104): Expandable structs may only be checked against types that are known to be record types. As with lambdas, the goal type must be known to be a sig, otherwise a constraint is generated and a meta returned. If the goal type is known to be a sig however, the e-struct is expanded as described in Figure 31.

$$\boxed{\underset{+}{\Xi};\underset{+}{\Gamma} \vdash \overrightarrow{\underset{+}{\Phi_C}} \rhd \overrightarrow{\underset{+}{f : U}} \rightsquigarrow \overrightarrow{\underset{-}{f := u}};\underset{-}{\Xi'}}$$

Struct expansion base

$$\frac{}{\Xi;\Gamma \vdash \epsilon \rhd \epsilon \rightsquigarrow \epsilon;\Xi} \tag{105}$$

Struct expansion cons

$$\frac{\begin{array}{c}\Xi;\Gamma \vdash \overrightarrow{\Phi_C}(f : U) \rightsquigarrow (u, \overrightarrow{\Phi'_C});\Xi' \\ \Xi',\Gamma \vdash \overrightarrow{\Phi'_C} \rhd \overrightarrow{f : U[u/f]} \rightsquigarrow \overrightarrow{f := u};\Xi''\end{array}}{\Xi;\Gamma \vdash \overrightarrow{\Phi_C} \rhd f : U, \overrightarrow{f : U} \rightsquigarrow f := u, \overrightarrow{f := u};\Xi''} \tag{106}$$

Match

$$\frac{\Xi;\Gamma \vdash e \leftleftarrows U \rightsquigarrow u;\Xi'}{\Xi;\Gamma \vdash (\Phi_C, \overrightarrow{\Phi_C})(f : U) \rightsquigarrow (u, \overrightarrow{\Phi_C});\Xi'} \quad [\Phi_C \in \{e, f := e\}] \tag{107}$$

No match

$$\frac{\Xi;\Gamma \vdash \_ \leftleftarrows U \rightsquigarrow u;\Xi'}{\Xi;\Gamma \vdash \overrightarrow{\Phi_C}(f : U) \rightsquigarrow (u, \overrightarrow{\Phi_C});\Xi'} \tag{108}$$

Figure 31 Algorithmic struct expansion

4. **Struct expansion** (105) - (108): The process of checking expandable records terminates successfully only when the empty $struct_e\{\}$ is matched against the empty $sig\{\}$. The recursive case depends on a matching judgment to decide whether to check a given value against the field type or to introduce a metavariable in the place of an omitted value (expansion). For the recursive call, the given value (or introduced meta) is substituted for all references to the current field name in the remainder of the goal type, which brings it into scope.

Checking expressions lacking any special checking rules

$$\frac{\Xi;\Gamma \vdash e \rightrightarrows U \rightsquigarrow u;\Xi' \quad \Xi';\Gamma \vdash u : U = T \rightsquigarrow t;\Xi''}{\Xi;\Gamma \vdash e \leftleftarrows T \rightsquigarrow t;\Xi''} \tag{109}$$

Equality by alpha conversion and reflexivity

$$\overline{\Xi;\Gamma \vdash u : (U = T) \rightsquigarrow u;\Xi} \quad [U \overset{\alpha}{=} T] \tag{110}$$

Generate equality constraint

$$\frac{\Xi;\Gamma \vdash genEqC(u : U = T) \rightsquigarrow X;\Xi'}{\Xi;\Gamma \vdash u : (U = T) \rightsquigarrow X;\Xi'} \tag{111}$$

Figure 32 Rules for checking general expressions by inference and type equality

5. **Checking an inferable expression** (109): The general expression checking rule is defined in terms of type inference and type equality. While Agda uses the subtyping relation to decide if the inferred type of an expression is permissible with respect to the type checked against, equality is sufficient for the purpose of this presentation. In order to check a general expression against a type T, first its type U is inferred, with an elaborated term u. The actual checking is then done through an equality test on T and U, which may block the usage of u with a fresh metavariable if the equality is not immediately decidable.

6. **Deciding type equality** (110): The first equality rule tries to solve the equality straight away using alpha conversion and reflexivity. It has precedence over the second (111), since both would otherwise be applicable. This rule ensures that constraints blocked on trivial type equalities, e.g. $Set = Set$, will not be generated.

7. **Generating an equality constraint** (111): If the types cannot be determined equal by structure alone, an equality constraint is generated. The purpose of the equality constraint is to block the usage of a term u with type U under a supposedly equal type T until the type equality $U = T$ is ensured. This is achieved by introducing a metavariable X of type T which takes the place of the term u until the constraint is resolved.

4.4.4 Rules for inferring the type of an expression

All inferable expressions are checked using inference and type equality. In the following list we explain the workings of each of the inference rules. In Figure 33 we present the inference rules for constructs that do not require any elaboration; the type will typically be the result of a look-up. Since contexts are assumed to be well-formed, and expressions well-scoped, these context look-ups will always succeed. In Figure 34 we present inference rules for function types, applications and record types; constructs which are elaborated recursively.

TYPE OF TYPES

$$\frac{}{\Xi; \Gamma \vdash \mathit{Set} \rightrightarrows \mathit{Set} \rightsquigarrow \mathit{Set}; \Xi} \tag{112}$$

CONSTANTS

$$\frac{\Xi; \Gamma \vdash \mathrm{lookup}(k, \Sigma) \rightsquigarrow T; \Xi'}{\Xi; \Gamma \vdash k \rightrightarrows T \rightsquigarrow k; \Xi'} \tag{113}$$

VARIABLES

$$\frac{\Xi; \Gamma \vdash \mathrm{lookup}(x, \Gamma) \rightsquigarrow T; \Xi'}{\Xi; \Gamma \vdash x \rightrightarrows T \rightsquigarrow x; \Xi'} \tag{114}$$

Figure 33 Type inference rules for constructs that require no elaboration

1. **Type of types** (112): The type of *Set* is inferred to be *Set*.
2. **Constants** (113): The type of a constant is inferred simply by looking up its declared type in Σ.
3. **Variables** (114): Just like constants, but the look-up is done in Γ.

Function types

$$\frac{\Xi;\Gamma \vdash D \Leftarrow \mathit{Set} \rightsquigarrow U;\Xi' \quad \Xi';\Gamma, x : U \vdash E \Leftarrow \mathit{Set} \rightsquigarrow V;\Xi''}{\Xi;\Gamma \vdash (x : D) \to E \Rightarrow \mathit{Set} \rightsquigarrow (x : U) \to V;\Xi''} \tag{115}$$

Applications

$$\frac{\Xi;\Gamma \vdash e_1 \Rightarrow (x : U) \to V \rightsquigarrow t;\Xi' \quad \Xi';\Gamma \vdash e_2 \Leftarrow U \rightsquigarrow u;\Xi''}{\Xi;\Gamma \vdash e_1\ e_2 \Rightarrow V[u/x] \rightsquigarrow t\ u;\Xi''} \tag{116}$$

Record type base

$$\frac{}{\Xi;\Gamma \vdash \mathit{sig}\{\} \Rightarrow \mathit{Set} \rightsquigarrow \mathit{sig}\{\};\Xi} \tag{117}$$

Record type cons

$$\frac{\Xi;\Gamma \vdash D \Leftarrow \mathit{Set} \rightsquigarrow U;\Xi' \quad \Xi';\Gamma, f : U \vdash \mathit{sig}\{\overrightarrow{f : D}\} \Leftarrow \mathit{Set} \rightsquigarrow \mathit{sig}\{\overrightarrow{f : U}\};\Xi''}{\Xi;\Gamma \vdash \mathit{sig}\{f : D, \overrightarrow{f : D}\} \Rightarrow \mathit{Set} \rightsquigarrow \mathit{sig}\{f : U, \overrightarrow{f : U}\};\Xi''} \tag{118}$$

Projections

$$\frac{\Xi;\Gamma \vdash e \Rightarrow \mathit{sig}\{\overrightarrow{f : U}\} \rightsquigarrow t;\Xi' \quad \Xi';\Gamma \vdash \mathrm{lookup}(f, \overrightarrow{f : U}) \rightsquigarrow U;\Xi''}{\Xi;\Gamma \vdash e.f \Rightarrow U[\overrightarrow{t.f/f}] \rightsquigarrow t.f;\Xi''} \tag{119}$$

Figure 34 Type inference rules for constructs requiring elaboration

4. **Function types** (115): As later types may depend on the values of earlier arguments, these variables must be added to the context before checking that later types have type *Set*. The argument types are thus checked in turn from left to right, updating the metavariable context as it goes along.

5. **Applications** (116): For the application rule we demand that the function e_1 to be applied to e_2 is inferable, and that the inferred type is indeed a function type. This means we can elaborate the argument by checking it against the domain of the function type ($e_2 \Leftarrow U$). The type of the whole application is the codomain V, with the elaborated argument u substituted for the variable x, bringing it into scope.

6. **Record types** (117) (118): As with function types, the record type is inferred to have type *Set* as long as all of the field types have type *Set* with the preceding fields in context.

7. **Projections** (119): For projections we demand that the target e is inferable, and that the inferred type is a record type. The type of the actual projection is then produced by looking up the type U of the matching field f, bringing it into scope by substituting all direct field references into projections.

5 Results

The Xiphi calculus is of limited practical usefulness, and the specified processing phases are quite naive in terms of prioritizing early failure. However, the elaboration process appears to work well for the examples we have studied and the deferred checking of lambdas in particular opens up for new investigations into how this behaviour should be handled in general. Furthermore, we show that the grouping imposed by the occurrence-restriction resolves the issues identified in the introduction, although with some impact on expressiveness.

5.1 Expressiveness

It is easy to show that expressiveness is not negatively affected by the restrictions imposed by the surface language. Due to the absence of implicit constructs standing alone it is not possible to construct expressions of the shapes, but the same semantics can be achieved by the insertion of dummies for bindings, types and arguments, to fulfill the grouping requirements (as shown in Figure 35).

Having to resort to using dummy bindings and things like the unit type and unit value would of course be highly annoying, but it should be possible to minimize the impact on usability by handling cases where these insertions can be safely automated. It is interesting to note that we would in that case have reached something akin to explicit-argument insertion. Additional issues related to expressiveness are mentioned in the discussion.

$\{x : T\} \to T$	$\{x : T\}\ (\_ : \top) \to T$
$\lambda\ \{x\} \to \ldots$	$\lambda\ \{x\}\ \_ \to \ldots$
$a\ \{x\}$	$a\ \{x\}\ ()$

Figure 35 Unavailable expression forms and their alternatives

5.2 Implementation of the $\lambda_{\Xi\Phi}$ calculus

In order to get a greater understanding of Xiphi, a Haskell implementation has been developed, the focus being direct correspondence to the type checking rules defined in this report.[8] It is readily apparent that many programs could be easily rejected directly in scope checking, while the current system happily lets them through to also pass elaboration, failing first by the identification of unsolvable constraints.

Although not intended to be a practical language in any sense, the implementation has proved useful when verifying the behaviour of the problem examples encoded in Xiphi.

[8] The code can be accessed at `https://github.com/jplloyd/xiphi`

5.3 Behaviour of the examples in $\lambda_{\Xi\Phi}$

To tie back to the problems used to define the shortcomings of hidden-lambda-insertion in the introductory section, we give their behaviour under type checking in the Xiphi calculus. The details of the derivations for the Ty and Lam examples can be found in Appendix A.2.

5.3.1 Ty example behaviour

To recap, the Ty example (Figure 7) demonstrates a case where a seemingly η-convertible definition fails to type check. We have encoded the definitions of both `fails` and `hacks` in Xiphi and it can shown that they, post-unification, will indeed both be convertible to the constant on which they are defined. This behaviour is of course expected as there are no longer any insertions of hidden arguments at the end of the defined application (and none can appear).

However, it is important to note that if we provide an explicit expansion for the second argument *manually*, the program will still fail to type check due to the very same constraint (lacking a general solution) as in Agda. We have only offset the behaviour for one level in this example; although one could make the argument that the expansion on the first level is more important, as it corresponds to a binding on a top-level definition where a naming the binding could help in understanding the expected input, the behaviour nevertheless remains inconsistent.

5.3.2 Vec example behaviour

The Vec example (Figure 8) is one sense the converse of the Ty example, as it demonstrates a case where a definition type checks while its corresponding η-like reduction fails to type check. Whereas the Ty example, when encoded in Xiphi, will type check for both reduced and expanded definitions, it can easily be seen that the Vec example will fail to type check for either definition. One only needs to consider that the grouping of quantifiers will be irrevocably different for the definitions of `vcons` and `cons`; in the former the type quantification will be grouped with the index quantification whereas for the latter they will be grouped with the explicit arguments of the element and vector respectively.

Just as in Agda though, there is a correspondence between the number of binders and the freedom of changing position and order of binds. Due to hidden-lambda insertion, Agda requires one less binder to type check permutations of implicits than in Xiphi, but the basic principle is the same. One could argue that the behaviour in Xiphi is preferable due to the necessary existence of a visible structure (a binder) which can be associated with the increased freedom for placement and permutation of hidden bindings in the declared type, but it is still somewhat arbitrary.

5.3.3 Lam example behaviour

The program corresponding to the Lam example (Figure 9) which we have encoded in Xiphi differs slightly in its behaviour compared to the original example, but contains the same fundamental problem; that of letting an inherently low-information structure (such as a lambda with a body containing only bound variables) determine the shape of a type.

While the original example used an implicit variable as an argument to a type function, locking that variable with an equality type, the behaviourally equivalent Lam example we encode in Xiphi bypass the type function and check on a type bound as an implicit argument directly. The slight difference in behaviour when the example is run in Agda is that the onus is placed on the argument to the equivalence class type to provide an argument compatible to the type inferred from the lambda (which leads to the same problem as in the original example, although the order is reversed).

When checking the lambda, the problem is paused and checking is continued with a placeholder after which the actual type will be locked by the value of the *eq* application. Upon resuming the elaboration the esig will be expanded against the sig of the newly determined function type. It must be noted that the procedure assumes that the lambda contains absolutely no information which might be used to infer its type, which may of course not always be the case.

In a case where closer inspection of the body of the lambda would indicate that the bound variable would have to be of a particular type in order for the body to type check, the desired behaviour might be to let the lambda steer the type checked against rather than the other way around.

5.3.4 Summary

The encodings of the problems show the same problematic behaviour when type checked in Agda, but fare fine when elaborated in the Xiphi calculus. Although we omit the final step of unification, the constraints produced by the examples are simple enough to tell that the whole type checking process works.

6 Discussion

Some limitations and considerations about the usefulness of $\lambda_{\Xi\Phi}$ as a system is discussed, and related to the current state of Agda. The section is concluded by a brief discussion on the practical feasibility of the solution.

6.1 Limitations on the expressiveness of $\lambda_{\Xi\Phi}$

While the proposed solution seemingly removes the need for hidden-lambda insertion, it does impose a number of limitations beyond the grouping-requirement, which – at least on the surface – affects the usefulness of the solution.

It should be noted that we do not allow type annotations in implicit binders (in lambda abstractions). Since lambdas in the type checking phase will only ever be (correctly) checked against function types, we consider the usefulness of having annotations in such positions to not outweigh the benefits of having simpler rules to reason about. In a real setting, it is very much possible that one would want typed lambdas for various reasons, be it for documentation or readability purposes, or simply to act as top-level definitions with built-in type declarations. A solution proposed by Ulf Norell would be to have two different kinds of lambdas, only one of which would be used as an argument. The question then becomes one of how best to distinguishing these lambdas, addressed further in the closing remarks.

Another rather more annoying limitation is the requirement that implicit bindings in lambda abstractions have to match the names of the corresponding binders in the types they are checked against. This is another case of making the rules easier at the cost of practical usability; such a requirement would be quite absurd in a real setting. However, this deficiency should be fairly easy to solve by simply introducing a construct in place of the field bindings which enables the optional "assignment" of a variable to a particular implicit bind (the functionality is present in Agda for function bindings, but not in lambda abstractions) and use indices with possible offsets to the closest assigned binding.

The most severe restriction from a practical standpoint is the omission of simple partial application of implicit arguments, due to the grouping imposed by the surface grammar. While it is technically possible to resolve this by inserting manual dummy arguments, the impact on usability is far too great in practice to make such a solution feasible. Some additional formalization of this has to be undertaken in order to make the solution fully usable.

6.2 Plausibility of integrating the solution into Agda

The proposed solution necessarily changes the semantics of some Agda programs containing implicit arguments, and as such legacy code might be negatively affected if the behaviour of Agda were to be changed. While changing the behaviour of some fundamental part of a programming language is always a risky thing to do, it would be interesting to assess the actual practical impact of this particular change, assuming the limitations of the syntax discussed previously can be resolved to satisfaction.

One difficulty is that apart from the standard library and presumably a few other places, the body of Agda code is scattered and largely private, with the exception of the work of enthusiasts accessible from public repositories. Nevertheless, if such an inventory of the occurrences of these niche cases were to be done, the result might indicate whether or not the benefits of the change would outweigh the problems of backwards-compatibility.

6.3 Closing remarks

While the problems we have prevented in this paper might be seen as niche or not too important, the primary use of Agda as a theorem prover means that every source of unreliability and unpredictability can be critical in a situation where a minor detail in an otherwise correct proof causes it to be rejected by the type checker.

Handling dependent function types with implicit arguments in a uniform way appears to resolve some of the described problems, but the trade-off with the current solution is that we don't allow a simple way of allowing for partial application of implicit arguments, which can be assumed to be a common use case in real programs. It seems quite likely that the benefits of handling problems of the type exemplified by the Ty example are outweighed by this limitation unless a way of handling partial application of implicits is added. When it comes to the Vec example, the behaviour is not fundamentally changed but rather shifted one step, and if considered a problem then that problem remains with this solution (with an offset of +1).

The different ways of handling lambda abstractions is something which Agda probably *could* benefit from however, and may be used in conjunction with something akin to the expandable signatures used in this work to provide deferred type checking of information-less lambdas. There are of course many problems which need to be investigated in order to realize a solution where lambdas are handled differently; what exactly constitutes an information-less lambda, how does one determine whether a lambda should or should not be inferred, and can this decision procedure itself have to be deferred if more information is provided at a later stage?

Altough many things remain unexplored, this work does at least provide some information about *one* path which may be taken to approach the problem.

References

[1] Gilles Barthe and Thierry Coquand. "An Introduction to Dependent Type Theory". In: *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*. London, UK, UK: Springer-Verlag, 2002, pp. 1–41. ISBN: 3-540-44044-5. URL: http://dl.acm.org/citation.cfm?id=647424.725800.

[2] Jacek Chrzaszcz. "Polymorphic subtyping without distributivity". In: vol. 1450. 1998, pp. 346–355. ISBN: 0302-9743.

[3] Warren D. Goldfarb. "The undecidability of the second-order unification problem". In: *Theoretical Computer Science* 13.2 (1981), pp. 225–230. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/0304-3975(81)90040-2. URL: http://www.sciencedirect.com/science/article/pii/0304397581900402.

[4] *Issue 1079: Hidden quantification*. https://code.google.com/p/agda/issues/detail?id=1079. Accessed: 2014-10-04.

[5] Connor McBride. *Faking It: Simulating Dependent Types in Haskell*. 2001.

[6] Ulf Norell. "Interactive Programming with Dependent Types". In: (2013). URL: http://dx.doi.org/10.1145/2500365.2500609.

[7] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Gteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007. URL: http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[8] Benjamin Pierce. *Types and programming languages*. Cambridge, Massachusetts: The MIT Press, 2002. ISBN: 0-262-16209-1.

[9] J. B. Wells. "Typability and type checking in System F are equivalent and undecidable". In: *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111–156.

A Derivations of the three defining problems Ty, Vec and Lam

This section details the type checking procedure of the three different programs exemplifying problematic behaviour for the current Agda type checker (see Section 1.2 for an overview). Ty and Vec exemplify counterintuitive behaviour of type checking function synonyms, while Lam demonstrates a problem with locking the type of an inferred lambda too quickly. First we give the derivations in Agda, making clear where the problems arise. Then we encode the same programs in our own calculus, Xiphi, and perform the derivations in its defined procedure to mark the difference.

A.1 Problem derivations in Agda

The type checking procedures presented herein share a common source of failure; the heuristic decision procedure of hidden lambda insertion. Further explanation of the current necessity of such a decision procedure, and what stages of elaboration that may demand this decision, is found in the practical setting of the stepped-through type checking derivations.

For an introduction to some of the notation used in the Agda derivations, please refer to Figure 36.

```
e ⇉ T   -- Check expression a against type T
e ⇇ T   -- Infer type T from expression a
T ≤ T'  -- Check that T is a subtype of T'
```

Figure 36 Notation for type checking operations

A.1.1 Eager or lazy? Two strategies for hidden-lambda insertion

To explain and illustrate the deficiencies of Agda's current type checker implementation, we introduce the concept of an *eager* strategy for inserting hidden lambdas compared to a *lazy* one. The eager strategy involves introducing hidden lambdas and pattern match binders as soon as the possibility arises, whereas lazy means deferring such action until there are no other ways of continuing elaboration and type checking.

It could be argued that these concepts correspond to a rather coarse analysis of the different possible designs of the decision heuristic, however they do provide a useful abstraction from the actual code of the type checker implementation. Additionally, we argue that using the current structure of implicit arguments in Agda makes it very hard, if not impossible, to design an efficient decision procedure for hidden lambda insertion that solves all problems presented. Therefore, we argue, it would be ill-conceived to search for a more fine-grained procedure for analyzing whether or not to insert hidden lambdas.

A.1.2 Ty: Excessive elaboration leading to unsolvable constraints

This first problem example was mentioned in the problem description (Section 1.2), and it constitutes a simplified version of a code segment from the bug report of a user (bug-ref).

Here we explain the exact process behind the failure to typecheck the alias definition from the example given in Figure 7:

```
fails : Ty
fails g = f g
```

1. **Traverse LHS and expand the context** First, when processing the LHS of the definition, the pattern variables are bound in the context for the RHS, associating them with the type information garnered from the declared type. Implicit arguments are inserted eagerly, meaning that whenever there is a way to add a binding for a hidden argument, given the shape of the type and the occurrences of visible arguments in the LHS, such a binding is added. In the case of `fails`, the first visible variable, `g`, will be bound to the first visible argument in `Ty`, associating it with the type `{b : Bool} →T b →T false`. Before this binding is done, however, we insert variables for the hidden arguments occurring before `g`, that being `{T}`. This results in the function-local context:

   ```
   Γ = T : Bool → Set, g : ({b : Bool} → T b → T false)
   ```

   which is used to check the definition of the RHS.

2. **Checking the definition** As at this point, the two arguments of `Ty` are bound on the LHS, we want to check the RHS against the remaining type, namely `T true`.

   The shape of the definition is an application, that of `f` to `g`, so we recursively check that application. When inferring the type of `f`, by a lookup in the context, fresh meta variables are inserted in the place of any hidden arguments occurring before the visible one, turning the application from `f g` into `f {_T} g`.

   By instantiating the implicit argument with the metavariable, the inferred type of `f {_T}` becomes:

   ```
   ({b : Bool} → _T b → _T false) → _T true
   ```

   and we must thus check the argument of `f {_T}`, that being `g`, against:

   ```
   ({b : Bool} → _T b → _T false).
   ```

   At this point we make a choice on how to handle the shape of the structure being considered.

3a. **(Eager) Insert a hidden abstraction** Since the type of the argument to `f` has a hidden argument, we insert a hidden abstraction corresponding to that argument and check the body (`g`) of the abstraction against the remainder of the type, `(_T b →_T false)`. This abstraction extends the context (available to the meta variables contained therein), with that binding (here named `b`). The shape of this type is different from the inferred

type of `g`, as the head is not implicit, and in order to achieve the same shape `g` has to be specialized by introducing a hidden argument ({_b}, for fresh metavariable _b) to which it can be applied. The shapes of the types now being equal, it remains to perform a unification for _b and _T such that the inferred type of the argument to `f` is a subtype of the declared argument type, the question being if:

```
(T _b → T false) ≤ (_T b → _T false)
```

which by structural subtyping turns into the smaller unification problems:

```
_T b ≤ T _b
```

and

```
T false ≤ _T false
```

of which _T b ≤ T _b lacks a unique solution. We can for example see that both of the unifiers

```
[_T := T, _b := b]
[_T := λ b → T false, _b := false]
```

provide valid solutions.

3b. (Lazy) Don't insert a hidden abstraction Neither hidden-abstraction-insertion, nor specialization of the argument `g`, is done prior to checking.

```
({b : Bool} → T b → T false)
≤
({b : Bool} → _T b → _T false)
```

where there is a unique solution, since the variable b is bound in both types, with the unifier

```
[ _T := λ b → T b]
```

This problem suggests that it would be beneficial to adopt a lazy strategy for hidden lambda insertion.

A.1.3 Vec: Function synonyms that maybe should not type check

This second problem example presents code which currently type checks, but would no longer do so should a lazy strategy be adopted in place of the eager one. This is however good, or at least tolerable, since it isn't obvious that it should type check in the first place. The definition being checked is that of the function `cons`, as seen in Figure 8.

1. Inspect LHS and build context We start by inspecting the LHS of the definition `cons a = vcons a`. Since `cons` is defined with pattern matching on just the first explicit argument (with type `A`), the implicit arguments preceding the explicit must be inserted through the process of elaboration. In the type definition of `cons`, this is just the type argument $\{A : Set\}$, which is inserted and added to the context.

```
cons a = vcons a ⇝ cons {A} a = vcons a
```

which adds `A : Set, a : A` to the context.

2. Choice of implicit-binding Since we now have a valid pattern matching definition on the LHS, we may continue by checking the RHS against the remaining type of `cons {A} a`, $\{$`n : ℕ`$\}$ →`Vec A n` →`Vec A (suc n)`. But we could just as well choose to include a match on $\{$`n`$\}$ on the LHS, leaving us to check `vcons a` ⇇`Vec A n` →`Vec A (suc n)`. This eager insertion of a hidden binder is how Agda currently proceeds.

3a. (Eager) Insertion We elaborate the pattern match to include $\{$`n : ℕ`$\}$:

```
cons {A} a = vcons a ⇝ cons {A} a {n} = vcons a
```

which adds the information that `(n : Nat)` to the context (at this point containing; `A : Set, a : A`)

4a. (Eager) Check RHS against remaining type of LHS We now want to check that the implementation of `cons` typechecks. Therefore we need to check the RHS against the remaining type of the LHS:

```
cons {A} a {n} : Vec A n → Vec A (suc n)

vcons a ⇇ Vec A n → Vec A (suc n)
```

5a. (Eager) Elaborate the application on RHS Since we have an application of `vcons` to `a`, we lookup the type of `vcons` which is

```
{A : Set} → {n : Nat} → A → Vec A n → Vec A (suc n).
```

(Note that the implicit parameter $\{$`A : Set`$\}$ is not mentioned explicitly in the definition of the `vcons` constructor since it is specified as a data type parameter, but it needs to be inserted wherever the constructor is used.)

We note that the definition of `vcons` takes two implicit arguments before the first explicit, and since it is applied to just an explicit argument `a`, we have to elaborate the term `vcons a` to match the defined type of `vcons`. This is done by adding metavariables in place of every implicit argument that is missing.

```
vcons a ⇝ vcons {_A} {_n} a
```

A metavariable may only be resolved by unifying it with a term that only contains variables that are stored in the context at the point of insertion of the metavariable. In this case the context consists of the type information; `(A : Set)`, `(a : A)` and `(n : ℕ)`, which were accumulated from the LHS. In order to see clearly the possible dependencies of each metavariable, the context could be included in the elaborated term as such:

```
vcons a ⇝ vcons {_A [A, a, n]} {_n [A, a, n]} a
```

Now we can infer the type of this elaborated term to be

```
vcons {_A} {_n} a ⇉ Vec _A _n → Vec _A (suc _n)
```

6a. (Eager) Unification Now we have to decide if the inferred type of `vcons a` can be used in place of the type that was expected by the original check procedure. Obviously this is true if the types are equal, but we can be a bit more permissive than that. By using the subtyping relation of Curry style System F, we may be able to produce meaningful substitutions in cases where pure equality would fail. In this case, even though equality is sufficient, we write the unification constraint using the subtyping relation (inferred type $\leq$ expected type):

```
Vec _A _n → Vec _A (suc _n)
≤
Vec A n → Vec A (suc n)
```

Since the topmost structures match (both are ordinary function types) we proceed by matching the domains. Functions are considered contravariant in their domain, which means the subtyping relation is inverted for the constraint

```
Vec A n ≤ Vec _A _n
```

The contravariance is not of importance here since the only solution to this equation is simply equality where `A` is substituted for `_A` and `n` for `_n`. Using these substitutions, the types of the codomains also become equal, which means we have successfully matched the inferred to the expected type, and therefore our implementation of `cons` by `cons a = vcons a` typechecks completely.

3b. (Lazy) No insertion Now what would have happened if we did not insert `{n}` eagerly in the pattern match of `vcons {A} a`, and had instead went on to checking the RHS right away?

4b. (Lazy) Check RHS against remaining type of LHS To recap, we elaborated the LHS `cons a` to `cons {A} a` and added `A : Set, a : A` to context. We move on to checking RHS against the remaining type of the LHS using this context.

```
cons {A} a : {n : ℕ} → Vec A n → Vec A (suc n)
```

```
vcons a ⇇ {n : ℕ} → Vec A n → Vec A (suc n)
```

5b. (Lazy) Elaborate application on RHS As in the eager case, we see an application and fetch the definition of `vcons`. Since `vcons` takes two implicit arguments before the first implicit one, we automatically insert metavariables in the missing places.

```
vcons a ⇝ vcons {_A} {_n} a
```

The key difference from the eager case is that the context does not contain `n`, since the LHS was not elaborated to include it. Including the context at the point of metavariable insertion makes this apparent:

```
vcons a ⇝ vcons {_A [A, a]} {_n [A, a]} a
```

These contexts are fixed, which means that whatever unification problem is produced by the next couple of steps, `_n` will never be unified with `n`, and thus type checking will fail. For completeness and clarity we outline the subsequent steps anyway.

After introducing the metavariables, we infer the remaining type to be

```
vcons {_A} {_n} a ⇉ Vec _A _n → Vec _A (suc _n)
```

But this type does not match the type checked for, which is

```
{n : ℕ} → Vec A n → Vec A (suc n)
```

Note that Agda would give up here, since our definition of a lazy strategy does not allow for post-insertion of hidden lambdas.

This particular example showcases something that reqires an eager strategy to type check. This shows there exists certain code, perhaps more useful than this example, that will cease to type check given a change in strategy. Weighing the pros and cons, one would probably suggest a shift to a lazy strategy, but the third example makes it clear why another approach might be necessary.

A.1.4 Lam: The effect of non-linear type information on lambdas

The presentation of the *Ty* and *Vec* problems suggests that the apparent counter-intuitive type checking behaviour possibly could be remedied by adopting a lazy strategy of inserting hidden lambdas. Unfortunately there are cases where neither an eager or lazy strategy is adequate, as showed by the following example.

The example here differs from the previous two in that it involves the use of equality types. Although the equality type defined here (see `_≡_` in Figure 9) is less general than the one used in Agda's standard library, in that it only quantifies over $\texttt{Set}_0$, they are functionally equivalent for this example. It exemplifies another situation where committing to one of two function shapes results in being unable to typecheck an otherwise correct definition. We consider the checking of the functions `works` and `fails`, as they are defined in Figure 9.

These two functions make use of the postulated functions `w` and `f` respectively, the types of which in turn depend on the definition of the type function `T`. `T` takes a boolean argument and returns one of two types. Here the significant part is that the shape of the type returned differs based on this argument; it contains an implicit argument in the case of `true`.

What is important to notice is that the function types of both `w` and `f` contain something using the ≡ symbol. These are instances of an equivalence type - a type indexed over two terms with the implication that those terms are the same. For the definition used here, the only constructor of these equivalence types is the nullary `refl`. The instances both state that `b = true` (for each of their scoped `b`), but what is important is the order in which these types appear in the definitions in relation to the computed type of `T`; prior to the `T b` in `w`, and after in `f`.

Although the scoped boolean argument `(b : Bool)` could (and since its value can be easily derived, should) be made implicit, we leave it as explicit to not confuse that implicit argument with the implicit which is the source of the problem. For all intents and purposes, it could be viewed as an implicit argument, for which we would of course omit the underscores in the function definitions (the derivations would be largely the same).

Checking "`works`"

1. Since there are no bindings on the LHS, the definition of `works` is checked against its declared type `Bool`.

2. The type of `w` is inferred (looked up in the global context), and the arguments are checked against it.

3. The first argument, the underscore, is handled by inserting a new metavariable `_b` of type `Bool` in it's place, and checking the remaining arguments against the instantiated type:
   `_b` ≡ `true` →`T` `_b` →`Bool`

4. Checking `refl` against `_b` ≡ `true`, we first infer the type of refl, which by definition is `_a≡_a` for a new metavariable `_a`. Testing that `_a≡_a` ≤ `_b` ≡ `true` yields unification constraints `_a = _b` and `_a = true`, and unification will lead to the substitution of `true` for `_b` in the remaining type.

5. Proceeding to check the lambda abstraction against `T true`, by computation the type checked against will be: `{A : Set} →A →A`. Since the type has an implicit argument, the lambda (which does not) is placed inside a hidden abstraction (hidden lambda insertion), which is then checked: `(λ{A : Set} λ x →x) ⇇ ({A : Set} →A →A)`

6. Extend the context with the binding for the implicit argument, `(A : Set)`, and check the body against the remaining type: `(λ x →x) ⇇ (A →A)`
   We begin by inferring the type of the lambda abstraction. A new metavariable `_X` for the type of the binding is added to the context, and we infer the type of the body, which is the bound variable `x`. The type of the lambda abstraction is thus inferred to `_X →_X`, and we need to check that:
   `_X →_X ≤ A →A`
   Unification solves this relation by substituting `A` for `_X` for the constraints generated by the subtyping.

7. Having checked the expressions of the arguments against the types declared in `w`, the type of the resulting expression is of type `Bool`, which matches the declared type of our definition, and the definition is successfully type checked.

Checking "`fails`"

Just like when checking the definition of `works`, we start by checking and similarly introduce a metavariable for `b`.

1. When checking the lambda abstraction, there is not enough information to determine the shape of the type `T _b` to check against, so we proceed by checking:
   `λ x →x ⇇ T _b` This leads to inferring of the type of the abstraction, which is done by introducing a metavariable `_X` for the binding, and inferring the body under a context extended by that binding. Since the definition only consists of the binding, the inferred type becomes `_X →_X` and for the definition to check we require that: `_X →_X ≤ T _b`. However, since there is not enough information to check that relation, the problem is put on hold and the remainder of the expression is checked.

2. When refl is checked, we get the exact same result as in the working definition, and since there is now new information about a metavariable we go back to the unsolved problem. The problem can now be instantiated:
   `_X →_X ≤ T true` , which by computation becomes:
   `_X →_X ≤ {A : Set} →A →A`

   At this point however, it is easy to note that no substitution of `_X` will yield the correct type, since we made an assumption about the shape of the type of the lambda when inferring the type of the abstraction.

Solving the *Lam* problem by modifying the handling of metavariables by allowing some sort of backtracking would unfortunately result in major computational overhead, since every instance of the metavariable would have to be located and have its context updated. We therefore conclude a reworking of the implicit argument handling is required to resolve this problem in a satisfying manner.

A.2 Problem derivations in Xiphi

By encoding the problematic Agda programs in Xiphi, we can show the type checking derivations and compare them to those of Agda. Only the Ty and Lam examples are derived, since the Vec example can trivially be shown to fail.

A.2.1 Notation and outline of the derivations

Type checking and inference in $\lambda_{\Xi\Phi}$ relies on an elaboration process that generates constraints which are later solved by unification. Since no unifier is defined, the values of the metavariables have been determined manually.

The derivations performed here follow the same structure. First an equivalent encoding of the problem is written in Agda, in a way which makes it syntactically compatible with our calculus. The Agda program is then translated to its equivalent representation in our lanugage, which is then used for the type checking derivation. To make the descriptions easier to follow, the elaboration and checking of postulates used in the main problem is left out, and the final type checked constructs are used immediately. The same goes for the type checked against in the main problems.

In these examples, an underscore to the left of a binding (e.g. $(\_ : \mathit{Set})$) will indicate that we don't care about the name of the binding, as it will not be used by the declarations that follows. It should not be confused with the underscore in the grammar, which will only ever occur in the place of an expression.

Due to their lenghty nature, the derivations are presented using a linear notation which indicate the existence of subtrees by using matching indices for between start and end points. Certain simple checks and inferences are placed on single lines and indicate the rules invoked as part of the transformation within parentheses. Sometimes multiple rules are invoked as part of a compact checking or inference, usually indicating equality by reflexivity or constraint-adding operations.

Inference uses double bracket ceiling and floor symbols, whereas checking uses the regular versions. Additionally, the paired arrows and $\rightsquigarrow$ used in the rules are present to further clarify what is inferred and what is checked. Contexts and context updates are not shown explicitly, as that would make it nearly impossible to follow the process, nor are constraint generations shown with the exception of the invoked rules. Each derivation is complemtented with the final output in terms of the elaborated term and the set of constraints generated as part of the process. For brevity, the final metastore is also omitted, as it is sufficient to know that the obvious instantiations indicated by the equality constraints are in the stored context, which can also be verified by following the derivation to the point of the generated meta.

A.2.2 Ty-derivation

This derivation uses the alternative Agda definition of the Lam example in Figure 37.

In this example, the Bool datatype only served as a means of distinction, and its ordinary semantics are not important, which is why we use *Set* instead in the encoding to avoid additional postulates. The common components, the postulate and type checked against, are first shown in the original surface encoding, as well as their scope checked and type checked counterparts in core and internal. The elaborations for both `works` and `fails` are then provided, showing that the former type checks directly, and the latter results in a trivially solvable equality constraint.

```
Ty = {T : (_ : Set) → Set} →
     (_ : {A : Set} → (_ : T A) → Set) → Set

postulate
  f : Ty

works : Ty
works = f

fails : Ty
fails = λ g → f g

hacks : Ty
hacks = λ g → f (λ {b} t → g {b} t)
```

Figure 37 Behaviourally equivalent $\lambda_{\Xi\Phi}$ -compatible Agda-implementation of the Ty-example

Postulates - Surface
$f : \{T : \{\}(\_ : Set) \to Set\}(\_ : \{A : Set\}(\_ : T\ \{\}\ A) \to Set) \to Set$

Type - Surface
$\{T : \{\}(\_ : Set) \to Set\}(\_ : \{A : Set\}(\_ : T\ \{\}\ A) \to Set) \to Set$

Postulates - Core
$f : (r_1 : sig\{T : (r_0 : sig\{\}) \to (x_0 : Set) \to Set\})$
$\quad \to (x_1 : (r_2 : sig\{A : Set\}) \to (x_2 : r_1.T\ struct_e\{\}\ r_2.A) \to Set) \to Set$

Type - Core
$(r_4 : sig\{T : (r_3 : sig\{\}) \to (x_3 : Set) \to Set\})$
$\to (x_4 : (r_5 : sig\{A : Set\}) \to (x_5 : r_4.T\ struct_e\{\}\ r_5.A) \to Set) \to Set$

Postulates - Internal
$f : (r_1 : sig\{T : (r_0 : sig\{\}) \to (x_0 : Set) \to Set\})$
$\quad \to (x_1 : (r_2 : sig\{A : Set\}) \to (x_2 : r_1.T\ struct\{\}\ r_2.A) \to Set) \to Set$

Type - Internal
$(r_4 : sig\{T : (r_3 : sig\{\}) \to (x_3 : Set) \to Set\})$
$\to (x_4 : (r_5 : sig\{A : Set\}) \to (x_5 : r_4.T\ struct\{\}\ r_5.A) \to Set) \to Set$

Derivation: Works

Expression - Surface/Core/Internal
f

Elaboration
$^{0}\lceil f \Leftarrow (r_4{:}sig\{\,T{:}(r_3{:}sig\{\,\}) \to (x_3{:}Set) \to Set\,\}) \to (x_4{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_4.T\ \ struct\{\,\}\ r_5.A) \to Set) \to Set \rceil^{0}$ (109)
$[\![f]\!]$ (113)$\Rightarrow (r_1{:}sig\{\,T{:}(r_0{:}sig\{\,\}) \to (x_0{:}Set) \to Set\,\}) \to (x_1{:}(r_2{:}sig\{\,A{:}Set\,\}) \to (x_2{:}r_1.T\ \ struct\{\,\}\ r_2.A) \to Set) \to Set \rightsquigarrow f$
$_{0}\lfloor$ (110)$\rightsquigarrow f$

Derivation: Fails

Expression - Surface
$\lambda\{\}g \to f\ \{\}\ g$

Expression - Core
$\lambda(r_6 : sig_e\{\}) \to \lambda(x_6 : \_) \to f\ \ struct_e\{\}\ x_6$

Elaboration
$^{0}\lceil \lambda(r_6{:}sig_e\{\}) \to \lambda(x_6{:}\_) \to f\ \ struct_e\{\}\ x_6 \Leftarrow (r_4{:}sig\{\,T{:}(r_3{:}sig\{\,\}) \to (x_3{:}Set) \to Set\,\}) \to (x_4{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_4.T\ \ struct\{\,\}\ r_5.A) \to Set) \to Set \rceil^{0}$ (101)
$^{1}\lceil \lambda(x_6{:}\_) \to f\ \ struct_e\{\}\ x_6 \Leftarrow (x_4{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_6.T\ \ struct\{\,\}\ r_5.A) \to Set) \to Set \rceil^{1}$ (101)
$^{2}\lceil f\ \ struct_e\{\}\ x_6 \Leftarrow Set \rceil^{2}$ (109)
$^{3}\lceil\!\lceil f\ \ struct_e\{\}\ x_6 \rceil\!\rceil^{3}$ (116)
$^{4}\lceil\!\lceil f\ \ struct_e\{\} \rceil\!\rceil^{4}$ (116)
$[\![f]\!]$ (113)$\Rightarrow (r_1{:}sig\{\,T{:}(r_0{:}sig\{\,\}) \to (x_0{:}Set) \to Set\,\}) \to$
$\quad (x_1{:}(r_2{:}sig\{\,A{:}Set\,\}) \to (x_2{:}r_1.T\ \ struct\{\,\}\ r_2.A) \to Set) \to Set \rightsquigarrow f$
$[struct_e\{\} \Leftarrow sig\{\,T{:}(r_0{:}sig\{\,\}) \to (x_0{:}Set) \to Set\,\}]$ (103)$\rightsquigarrow struct\{\,T{:=}X_0\,\}$
$_{4}\lfloor\!\lfloor \Rightarrow (x_1{:}(r_2{:}sig\{\,A{:}Set\,\}) \to (x_2{:}X_0\ \ struct\{\,\}\ r_2.A) \to Set) \to Set \rightsquigarrow f\ \ struct\{\,T{:=}X_0\,\}$
$^{6}\lceil x_6 \Leftarrow (r_2{:}sig\{\,A{:}Set\,\}) \to (x_2{:}X_0\ \ struct\{\,\}\ r_2.A) \to Set \rceil^{6}$ (109)
$[\![x_6]\!]$ (114)$\Rightarrow (r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_6.T\ \ struct\{\,\}\ r_5.A) \to Set \rightsquigarrow x_6$
$_{6}\lfloor$ (98)$\rightsquigarrow X_1$
$_{3}\lfloor\!\lfloor \Rightarrow Set \rightsquigarrow f\ \ struct\{\,T{:=}X_0\,\}\ X_1$
$_{2}\lfloor$ (110)$\rightsquigarrow f\ \ struct\{\,T{:=}X_0\,\}\ X_1$
$_{1}\lfloor \rightsquigarrow \lambda(x_6{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_6.T\ \ struct\{\,\}\ r_5.A) \to Set) \to f\ \ struct\{\,T{:=}X_0\,\}\ X_1$
$_{0}\lfloor \rightsquigarrow \lambda(r_6{:}sig\{\,T{:}(r_3{:}sig\{\,\}) \to (x_3{:}Set) \to Set\,\}) \to$
$\quad \lambda(x_6{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_6.T\ \ struct\{\,\}\ r_5.A) \to Set) \to f\ \ struct\{\,T{:=}X_0\,\}\ X_1$

Expression - Internal
$\lambda(r_6{:}sig\{\,T{:}(r_3{:}sig\{\,\}) \to (x_3{:}Set) \to Set\,\}) \to \lambda(x_6{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_6.T\ \ struct\{\,\}\ r_5.A) \to Set) \to$
$f\ \ struct\{\,T{:=}X_0\,\}\ X_1$

Constraints
$(x_6{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_6.T\ \ struct\{\,\}\ r_5.A) \to Set$
$=$
$(r_2{:}sig\{\,A{:}Set\,\}) \to (x_2{:}X_0\ \ struct\{\,\}\ r_2.A) \to Set \dagger X_1)$
$[x_6{:}(r_5{:}sig\{\,A{:}Set\,\}) \to (x_5{:}r_6.T\ \ struct\{\,\}\ r_5.A) \to Set,\ r_6{:}sig\{\,T{:}(r_3{:}sig\{\,\}) \to (x_3{:}Set) \to Set\,\}]$

A.2.3 Lam-derivation

The equivalent Agda definition of the Lam example (Figure 38) is used in this derivation.

Since the basic principle behind deferring lambda checking is that the type may be determined after the natural checking of the lambda, the example is simplified to a bare minimum, letting the type itself be the implicit argument and using an equivalence-class data structure to encode the way by which the argument is locked. Both `works` (linear) and `fails` (nonlin) now produce symmetrical constraints, which upon resolution resumes the checking of the lambdas, which can now be guided by the type, resulting in the eventual correct instantiation of the sig.

```
postulate
  Eq : (_ : Set1) → Set1
  eq : {T : Set1} → Eq T
  w : {T : Set1} → (_ : Eq T) → (_ : T) → Set
  f : {T : Set1} → (_ : T) → (_ : Eq T) → Set

linear : Set
linear = w (eq ({A : Set} → (a : A) → a)) (λ x → x)

nonlin : Set
nonlin = f (λ x → x ) (eq ({A : Set} → (a : A) → a))
```

Figure 38 Behaviourally equivalent definition of the Lam example

Postulates - Surface
$Eq : \{\}(\_ : Set) \to Set$
$eq : \{\}(T : Set) \to Eq\ \{\}\ T$
$w : \{T : Set\}(\_ : Eq\ \{\}\ T) \to \{\}(\_ : T) \to Set$
$f : \{T : Set\}(\_ : T) \to \{\}(\_ : Eq\ \{\}\ T) \to Set$

Postulates - Core
$Eq : (r_0 : sig\{\}) \to (x_0 : Set) \to Set$
$eq : (r_1 : sig\{\}) \to (x_1 : Set) \to Eq\ struct_e\{\}\ x_1$
$w : (r_2 : sig\{T : Set\}) \to (x_2 : Eq\ struct_e\{\}\ r_2.T)$
$\quad \to (r_3 : sig\{\}) \to (x_3 : r_2.T) \to Set$
$f : (r_4 : sig\{T : Set\}) \to (x_4 : r_4.T) \to (r_5 : sig\{\})$
$\quad \to (x_5 : Eq\ struct_e\{\}\ r_4.T) \to Set$

Postulates - Internal
$Eq : (r_0 : sig\{\}) \to (x_0 : Set) \to Set$
$eq : (r_1 : sig\{\}) \to (x_1 : Set) \to Eq\ struct\{\}\ x_1$
$w : (r_2 : sig\{T : Set\}) \to (x_2 : Eq\ struct\{\}\ r_2.T)$
$\quad \to (r_3 : sig\{\}) \to (x_3 : r_2.T) \to Set$
$f : (r_4 : sig\{T : Set\}) \to (x_4 : r_4.T)$
$\quad \to (r_5 : sig\{\}) \to (x_5 : Eq\ struct\{\}\ r_4.T) \to Set$

Type - Surface/Core/Internal
Set

Derivation: Linear

Expression - Surface

$w\ \{\}\ (eq\ \{\}\ (\{A : Set\}(\_ : A) \to A))\ \{\}\ (\lambda\{\}x \to x)$

Expression - Core

$w\ struct_e\{\}\ (eq\ struct_e\{\}\ ((r_6 : sig\{A : Set\}) \to (x_6 : r_6.A) \to r_6.A))$
$\quad struct_e\{\}\ (\lambda(r_7 : sig_e\{\}) \to \lambda(x_7 : \_) \to x_7)$

Elaboration

$^{0}\lceil w\ struct_e\{\}\ (eq\ struct_e\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A))\ struct_e\{\}\ (\lambda(r_7{:}sig_e\{\})\to\lambda(x_7{:}\_)\to x_7)\Leftarrow Set\rceil^{0}$ (109)
$^{1}\lVert w\ struct_e\{\}\ (eq\ struct_e\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A))\ struct_e\{\}\ (\lambda(r_7{:}sig_e\{\})\to\lambda(x_7{:}\_)\to x_7)\rVert^{1}$ (116)
$^{2}\lVert w\ struct_e\{\}\ (eq\ struct_e\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A))\ struct_e\{\}\rVert^{2}$ (116)
$^{3}\lVert w\ struct_e\{\}\ (eq\ struct_e\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A))\rVert^{3}$ (116)
$^{4}\lVert w\ struct_e\{\}\rVert^{4}$ (116)
$[\![w]\!]$ (113)$\rightrightarrows(r_2{:}sig\{T{:}Set\})\to(x_2{:}Eq\ struct\{\}\ r_2.T)\to(r_3{:}sig\{\})\to(x_3{:}r_2.T)\to Set\rightsquigarrow w$
$[struct_e\{\}\Leftarrow sig\{T{:}Set\}]$ (103)$\rightsquigarrow struct\{T{:=}X_0\}$
$_{4}\lVert\rightrightarrows(x_2{:}Eq\ struct\{\}\ X_0)\to(r_3{:}sig\{\})\to(x_3{:}X_0)\to Set\rightsquigarrow w\ struct\{T{:=}X_0\}$
$^{6}\lceil eq\ struct_e\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A)\Leftarrow Eq\ struct\{\}\ X_0\rceil^{6}$ (109)
$^{7}\lVert eq\ struct_e\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A)\rVert^{7}$ (116)
$^{8}\lVert eq\ struct_e\{\}\rVert^{8}$ (116)
$[\![eq]\!]$ (113)$\rightrightarrows(r_1{:}sig\{\})\to(x_1{:}Set)\to Eq\ struct\{\}\ x_1\rightsquigarrow eq$
$[struct_e\{\}\Leftarrow sig\{\}]$ (103)$\rightsquigarrow struct\{\}$
$_{8}\lVert\rightrightarrows(x_1{:}Set)\to Eq\ struct\{\}\ x_1\rightsquigarrow eq\ struct\{\}$
$^{10}\lceil(r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A\Leftarrow Set\rceil^{10}$ (109)
$^{11}\lVert(r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A\rVert^{11}$ (115)
$^{12}\lceil sig\{A{:}Set\}\Leftarrow Set\rceil^{12}$ (109)
$^{13}\lVert sig\{A{:}Set\}\rVert^{13}$ (118)
$[Set\Leftarrow Set]$ (109) (112) (110)$\rightsquigarrow Set$
$^{15}\lceil sig\{\}\Leftarrow Set\rceil^{15}$ (109)
$[\![sig\{\}]\!]$ (117)$\rightrightarrows Set\rightsquigarrow sig\{\}$
$_{15}\lfloor$ (110)$\rightsquigarrow sig\{\}$
$_{13}\lVert\rightrightarrows Set\rightsquigarrow sig\{A{:}Set\}$
$_{12}\lfloor$ (110)$\rightsquigarrow sig\{A{:}Set\}$
$^{16}\lceil(x_6{:}r_6.A)\to r_6.A\Leftarrow Set\rceil^{16}$ (109)
$^{17}\lVert(x_6{:}r_6.A)\to r_6.A\rVert^{17}$ (115)
$^{18}\lceil r_6.A\Leftarrow Set\rceil^{18}$ (109)
$^{19}\lVert r_6.A\rVert^{19}$ (119)
$[\![r_6]\!]$ (114)$\rightrightarrows sig\{A{:}Set\}\rightsquigarrow r_6$
$_{19}\lVert\rightrightarrows Set\rightsquigarrow r_6.A$
$_{18}\lfloor$ (110)$\rightsquigarrow r_6.A$
$^{20}\lceil r_6.A\Leftarrow Set\rceil^{20}$ (109)
$^{21}\lVert r_6.A\rVert^{21}$ (119)
$[\![r_6]\!]$ (114)$\rightrightarrows sig\{A{:}Set\}\rightsquigarrow r_6$
$_{21}\lVert\rightrightarrows Set\rightsquigarrow r_6.A$
$_{20}\lfloor$ (110)$\rightsquigarrow r_6.A$
$_{17}\lVert\rightrightarrows Set\rightsquigarrow(x_6{:}r_6.A)\to r_6.A$
$_{16}\lfloor$ (110)$\rightsquigarrow(x_6{:}r_6.A)\to r_6.A$
$_{11}\lVert\rightrightarrows Set\rightsquigarrow(r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A$
$_{10}\lfloor$ (110)$\rightsquigarrow(r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A$
$_{7}\lVert\rightrightarrows Eq\ struct\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A)\rightsquigarrow eq\ struct\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A)$
$_{6}\lfloor$ (98)$\rightsquigarrow X_1$
$_{3}\lVert\rightrightarrows(r_3{:}sig\{\})\to(x_3{:}X_0)\to Set\rightsquigarrow w\ struct\{T{:=}X_0\}\ X_1$
$[struct_e\{\}\Leftarrow sig\{\}]$ (103)$\rightsquigarrow struct\{\}$
$_{2}\lVert\rightrightarrows(x_3{:}X_0)\to Set\rightsquigarrow w\ struct\{T{:=}X_0\}\ X_1\ struct\{\}$
$[\![\lambda(r_7{:}sig_e\{\})\to\lambda(x_7{:}\_)\to x_7]\!]$ (102) (97)$\rightrightarrows Set\rightsquigarrow X_2$
$_{1}\lVert\rightrightarrows Set\rightsquigarrow w\ struct\{T{:=}X_0\}\ X_1\ struct\{\}\ X_2$
$_{0}\lfloor$ (110)$\rightsquigarrow w\ struct\{T{:=}X_0\}\ X_1\ struct\{\}\ X_2$

Expression - Internal

$w\ struct\{T{:=}X_0\}\ X_1\ struct\{\}\ X_2$

Constraints

$(\lambda(r_7{:}sig_e\{\})\to\lambda(x_7{:}\_)\to x_7\Leftarrow X_0\dagger X_2)[]$

$(eq\ struct\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A){:}Eq\ struct\{\}\ ((r_6{:}sig\{A{:}Set\})\to(x_6{:}r_6.A)\to r_6.A){=}Eq\ struct\{\}\ X_0\dagger X_1)[]$

Derivation: Nonlinear

Expression - Surface

$f\ \{\}\ (\lambda\{\}x \to x)\ \{\}\ (eq\ \{\}\ (\{A : Set\}(\_ : A) \to A))$

Expression - Core

$f\ struct_e\{\}\ (\lambda(r_6 : sig_e\{\}) \to \lambda(x_6 : \_) \to x_6)$
$\quad struct_e\{\}\ (eq\ struct_e\{\}\ ((r_7 : sig\{A : Set\}) \to (x_7 : r_7.A) \to r_7.A))$

Elaboration

$^{0}\lceil f\ struct_e\{\}\ (\lambda(r_6{:}sig_e\{\}){\to}\lambda(x_6{:}\_){\to}x_6)\ struct_e\{\}\ (eq\ struct_e\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A)){\Leftarrow}Set\rceil^{0}$ (109)
$^{1}\lVert f\ struct_e\{\}\ (\lambda(r_6{:}sig_e\{\}){\to}\lambda(x_6{:}\_){\to}x_6)\ struct_e\{\}\ (eq\ struct_e\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A))\rVert^{1}$ (116)
$^{2}\lVert f\ struct_e\{\}\ (\lambda(r_6{:}sig_e\{\}){\to}\lambda(x_6{:}\_){\to}x_6)\ struct_e\{\}\rVert^{2}$ (116)
$^{3}\lVert f\ struct_e\{\}\ (\lambda(r_6{:}sig_e\{\}){\to}\lambda(x_6{:}\_){\to}x_6)\rVert^{3}$ (116)
$^{4}\lVert f\ struct_e\{\}\rVert^{4}$ (116)
$[\![f]\!]$ (113)$\Rightarrow(r_4{:}sig\{T{:}Set\}){\to}(x_4{:}r_4.T){\to}(r_5{:}sig\{\}){\to}(x_5{:}Eq\ struct\{\}\ r_4.T){\to}Set{\rightsquigarrow}f$
$[struct_e\{\}{\Leftarrow}sig\{T{:}Set\}]$ (103)$\rightsquigarrow struct\{T{:=}X_0\}$
$_{4}\lVert{\Rightarrow}(x_4{:}X_0){\to}(r_5{:}sig\{\}){\to}(x_5{:}Eq\ struct\{\}\ X_0){\to}Set{\rightsquigarrow}f\ struct\{T{:=}X_0\}$
$[\![\lambda(r_6{:}sig_e\{\}){\to}\lambda(x_6{:}\_){\to}x_6]\!]$ (102) (97)$\Rightarrow Set{\rightsquigarrow}X_1$
$_{3}\lVert{\Rightarrow}(r_5{:}sig\{\}){\to}(x_5{:}Eq\ struct\{\}\ X_0){\to}Set{\rightsquigarrow}f\ struct\{T{:=}X_0\}\ X_1$
$[struct_e\{\}{\Leftarrow}sig\{\}]$ (103)$\rightsquigarrow struct\{\}$
$_{2}\lVert{\Rightarrow}(x_5{:}Eq\ struct\{\}\ X_0){\to}Set{\rightsquigarrow}f\ struct\{T{:=}X_0\}\ X_1\ struct\{\}$
$^{7}\lceil eq\ struct_e\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A){\Leftarrow}Eq\ struct\{\}\ X_0\rceil^{7}$ (109)
$^{8}\lVert eq\ struct_e\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A)\rVert^{8}$ (116)
$^{9}\lVert eq\ struct_e\{\}\rVert^{9}$ (116)
$[\![eq]\!]$ (113)$\Rightarrow(r_1{:}sig\{\}){\to}(x_1{:}Set){\to}Eq\ struct\{\}\ x_1{\rightsquigarrow}eq$
$[struct_e\{\}{\Leftarrow}sig\{\}]$ (103)$\rightsquigarrow struct\{\}$
$_{9}\lVert{\Rightarrow}(x_1{:}Set){\to}Eq\ struct\{\}\ x_1{\rightsquigarrow}eq\ struct\{\}$
$^{11}\lceil(r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A{\Leftarrow}Set\rceil^{11}$ (109)
$^{12}\lVert(r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A\rVert^{12}$ (115)
$^{13}\lceil sig\{A{:}Set\}{\Leftarrow}Set\rceil^{13}$ (109)
$^{14}\lVert sig\{A{:}Set\}\rVert^{14}$ (118)
$[Set{\Leftarrow}Set]$ (109) (112) (110)$\rightsquigarrow Set$
$^{16}\lceil sig\{\}{\Leftarrow}Set\rceil^{16}$ (109)
$[\![sig\{\}]\!]$ (117)$\Rightarrow Set{\rightsquigarrow}sig\{\}$
$_{16}\lfloor$ (110)$\rightsquigarrow sig\{\}$
$_{14}\lVert{\Rightarrow}Set{\rightsquigarrow}sig\{A{:}Set\}$
$_{13}\lfloor$ (110)$\rightsquigarrow sig\{A{:}Set\}$
$^{17}\lceil(x_7{:}r_7.A){\to}r_7.A{\Leftarrow}Set\rceil^{17}$ (109)
$^{18}\lVert(x_7{:}r_7.A){\to}r_7.A\rVert^{18}$ (115)
$^{19}\lceil r_7.A{\Leftarrow}Set\rceil^{19}$ (109)
$^{20}\lVert r_7.A\rVert^{20}$ (119)
$[\![r_7]\!]$ (114)$\Rightarrow sig\{A{:}Set\}{\rightsquigarrow}r_7$
$_{20}\lVert{\Rightarrow}Set{\rightsquigarrow}r_7.A$
$_{19}\lfloor$ (110)$\rightsquigarrow r_7.A$
$^{21}\lceil r_7.A{\Leftarrow}Set\rceil^{21}$ (109)
$^{22}\lVert r_7.A\rVert^{22}$ (119)
$[\![r_7]\!]$ (114)$\Rightarrow sig\{A{:}Set\}{\rightsquigarrow}r_7$
$_{22}\lVert{\Rightarrow}Set{\rightsquigarrow}r_7.A$
$_{21}\lfloor$ (110)$\rightsquigarrow r_7.A$
$_{18}\lVert{\Rightarrow}Set{\rightsquigarrow}(x_7{:}r_7.A){\to}r_7.A$
$_{17}\lfloor$ (110)$\rightsquigarrow(x_7{:}r_7.A){\to}r_7.A$
$_{12}\lVert{\Rightarrow}Set{\rightsquigarrow}(r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A$
$_{11}\lfloor$ (110)$\rightsquigarrow(r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A$
$_{8}\lVert{\Rightarrow}Eq\ struct\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A){\rightsquigarrow}eq\ struct\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A)$
$_{7}\lfloor$ (98)$\rightsquigarrow X_2$
$_{1}\lVert{\Rightarrow}Set{\rightsquigarrow}f\ struct\{T{:=}X_0\}\ X_1\ struct\{\}\ X_2$
$_{0}\lfloor$ (110)$\rightsquigarrow f\ struct\{T{:=}X_0\}\ X_1\ struct\{\}\ X_2$

Expression - Internal

$f\ struct\{T{:=}X_0\}\ X_1\ struct\{\}\ X_2$

Constraints

$(eq\ struct\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A){:}Eq\ struct\{\}\ ((r_7{:}sig\{A{:}Set\}){\to}(x_7{:}r_7.A){\to}r_7.A){=}Eq\ struct\{\}\ X_0{\dagger}X_2)[]$

$(\lambda(r_6{:}sig_e\{\}){\to}\lambda(x_6{:}\_){\to}x_6{\Leftarrow}X_0{\dagger}X_1)[]$

B Miscellaneous operations and properties

This section presents the details of some operations and properties which did not fit into the main report. First, well-formedness for contexts is treated, then well-formedness and computation of substitutions, and then equality of types and terms. This gives a more detailed view of what is assumed of the mentioned structures as we transform them using our rules for type checking.

B.1 Well-formedness of contexts

The rules for well-formedness of contexts basically define what constructs are to be stored within them. In the typing rules for the internal language, we implicitly assume that all contexts used in the premises are well-formed, and make sure that well-formedness is retained for all rules. The formal rules for well-formedness of the different contexts used in type checking are presented in Figures 39 (Σ), 40 (Γ), and 41 (Ξ).

Well-formedness for Σ in means that it only contains valid typings, and the only way to enforce this is to see that all types are proper, i.e., that they have type Set. The rules are defined recursively, with the base case (120) stating that an empty Σ is always well-formed, and with the recursive step (121) stating that the last added binding has a proper type (see Figure 19).

Well-formedness for Γ have rules which are very similar to the rules for Σ. The only difference is that Γ depends on the well-formedness of Ξ, since the types from the variable bindings in Γ are allowed to contain metavariables from Ξ. The base case (122) thus must state the well-formedness of Ξ, and the recursive case (123) must assume and propagate Ξ.

Well-formedness for Ξ may contain elements of three different shapes. The base case (124) is straightforward with an empty context always being well-formed. The first recursive case (125) introduces a metavariable X with type T along with a well-formed context Γ wherein T is a proper type. The second (126) adds an equality constraint, where u has type U, the meta X has type T, but it may not yet have been decided whether $U = T$ or not. The third (127) adds a checking constraint, where the expression e must be well-scoped, and the meta X has to be of type T, although the type of e might not match T.

Well-formedness of Σ

$$\boxed{\vdash \Sigma}$$

Empty context

$$\frac{}{\vdash \cdot} \tag{120}$$

Cons with typing

$$\frac{\vdash \Sigma \qquad \Sigma \vdash T : Set}{\vdash \Sigma, k : T} \tag{121}$$

Figure 39 Well-formedness of the global context.

Well-formedness of Γ

$$\boxed{\Xi \vdash \Gamma}$$

Empty variable context

$$\frac{\vdash \Xi}{\Xi \vdash \cdot} \tag{122}$$

Variable context cons

$$\frac{\Xi \vdash \Gamma \qquad \Gamma \vdash T : \mathit{Set}}{\Xi \vdash \Gamma, x : T} \tag{123}$$

Figure 40 Well-formedness of variable contexts

Well-formedness of Ξ

$$\boxed{\vdash \Xi}$$

Empty context

$$\frac{}{\vdash \cdot} \tag{124}$$

Metavariable introduction

$$\frac{\Xi; \Gamma \vdash T : \mathit{Set}}{\vdash \Xi, (X : T)[\Gamma]} \tag{125}$$

Equality constraint

$$\frac{\Xi; \Gamma \vdash u : U \quad \Xi; \Gamma \vdash X : T}{\vdash \Xi, (u : U = T \dagger X)[\Gamma]} \tag{126}$$

Checking constraint

$$\frac{\Xi; \Gamma \vdash X : T}{\vdash \Xi, (e \Leftarrow T \dagger X)[\Gamma]} \tag{127}$$

Figure 41 Well-formedness of metavariable contexts

B.2 Well-formedness of substitutions

The rule that defines well-formedness for substitutions is shown in Figure 42. It is defined on a vector of substitutions to be compatible with the vector of substitutions carried around by metavariables.

$$\frac{\Xi; \Gamma \vdash \overrightarrow{u : U} \qquad \Xi; \Gamma, \overrightarrow{x : U} \vdash t : T}{\Xi; \Gamma \vdash t[\overrightarrow{u/x}] : T[\overrightarrow{u/x}]} \tag{128}$$

Figure 42 Rule for well-formed substitutions

The rule uses shorthand vector notation for well-typedness of all terms in the range of the substitution, as well as for putting all variables from the substitution domain into context. Well-formedness for a vector of substitutions entails the provision of a term for every free variable in t which is not contained in Γ, meaning that the computation of all substitutions contained in the vector brings t into the scope of Γ.

B.3 Computing substitutions on terms

The rules for computing substitutions are collected into the recursively defined σ-function presented in Figure 43.

$$\begin{aligned}
((x : U) \to V)[\sigma] &= (x : U[\sigma]) \to V[\sigma] & (t.f)[\sigma] &= t[\sigma].f \\
(\lambda(x : U) \to v)[\sigma] &= \lambda(x : U[\sigma]) \to v[\sigma] & X[\overrightarrow{\sigma}][\sigma] &= X[\overrightarrow{\sigma}, \sigma] \\
(t\ u)[\sigma] &= t[\sigma]\ u[\sigma] & x[u/x] &= u \\
sig\{\overrightarrow{f : U}\}[\sigma] &= sig\{\overrightarrow{f : U[\sigma]}\} & y[u/x] &= y \\
struct\{\overrightarrow{f := u}\}[\sigma] &= struct\{\overrightarrow{f := u[\sigma]}\} & t[\sigma] &= t
\end{aligned}$$

Figure 43 Definition of the σ function which computes substitutions.

If the term on which the substitution is to be computed is of some recursive structure, the substitution is propagated down using distributive rules. When reaching a matching variable the actual substitution will be performed, whereas any other non-recursive term will simply discard the substitution.

Since substitutions cannot be computed for metavariables, they are simply stored to be computed later when a valid instantiation has been found during unification. There is no need to avoid variable capturing since the scope checker has transformed all local bindings and variables into globally unique counterparts. This also has the consequence that when substitutions are computed on instantiated metas, they can simply be processed one by one, as long as every stored substitution is computed.

B.4 Equality of types and terms

This section presents declarative rules for structural equality of terms, mainly focused on comparing the shapes of terms but also including eta- and alpha-conversion. The rules are only concerned with comparing terms that have been shown to have the same type. Well-formedness is assumed for contexts.

An actual algorithm for deciding term equality as specified by these rules could be achieved roughly as follows: first reduce both terms to weak head normal form , then compare the heads; if the heads are not equal, the process fails, otherwise repeat the process for all subterms. The compared terms are considered equal when this process terminates without failing.

Reduction to weak head normal form may be done by applying any matching reduction to a term, repeating this on the result until no more reductions can be made (hopefully given some guarantee that this procedure will terminate). An algorithm would also have to take into account the possibility of terms being equal as specified by the rules α and η conversion. In the case of α-equality a practical solution would employ De-Bruijn indices or similar.

Equality abides by the well known laws of reflexivity, transitivity and symmetry. It is also possible to show equality by reducing and converting terms using the various rules supplied and described in the next subsection. For more details, see Figure 44.

General shape for term equality

$$\boxed{\Xi; \Gamma \vdash t1 = t2}$$

Eq law: Reflexivity

$$\frac{}{\Xi; \Gamma \vdash t = t} \quad (129)$$

Eq law: Transitivity

$$\frac{\Xi; \Gamma \vdash t1 = t2 \qquad \Xi; \Gamma \vdash t2 = t3}{\Xi; \Gamma \vdash t1 = t3} \quad (130)$$

Eq law: Symmetry

$$\frac{\Xi; \Gamma \vdash t1 = t2}{\Xi; \Gamma \vdash t2 = t1} \quad (131)$$

Eq from reduction or conversion

$$\frac{\Xi; \Gamma \vdash t \stackrel{*}{=} t'}{\Xi; \Gamma \vdash t = t'} \quad [* \in \{\alpha, \beta, \eta\}] \quad (132)$$

Figure 44 Term equality

B.5 Term reductions and conversions

Type checking dependent types requires the computation of functions. The simple scheme we have used in this thesis defers all computation of functions to the unification phase. These are the computational rules available to the unifier to reduce terms to weak head normal form for equality comparisons.

In this presentation, conversions differ from reductions in that they are naturally reversable processes, thus in some sense non-directional, whereas reductions are computed with the purpose of producing a normalized value from some compound structure.

B.5.1 Alpha conversion

Alpha conversion (Figure 45) enables function types and abstractions to be considered equal even though the names of their local variables differ. Equality modulo alpha conversion may be achieved efficiently through the use of de Bruijn indices, but in our case renaming using substitution will do.

FUNCTION TYPE

$$\frac{\Xi;\Gamma \vdash U1 = U2 \qquad \Xi;\Gamma, x1 : U1 \vdash V1 = (V2[{}^{x1}/_{x2}])}{\Xi;\Gamma \vdash (x1 : U1) \to V1 \overset{\alpha}{=} (x2 : U2) \to V2} \tag{133}$$

LAMBDA ABSTRACTION

$$\frac{\Xi;\Gamma \vdash U1 = U2 \qquad \Xi;\Gamma, x1 : U1 \vdash v1 = (v2[{}^{x1}/_{x2}])}{\Xi;\Gamma \vdash (x1 : U1) \to v1 \overset{\alpha}{=} (x2 : U2) \to v2} \tag{134}$$

Figure 45 Alpha conversion of function types and lambda abstractions

B.5.2 Beta reduction

Beta reduction (Figure 46) is used to compute applications of lambdas to arguments by making use of substitutions, and also computing projections by looking up the value of a named field in a given struct.

APPLYING A FUNCTION TO AN ARGUMENT

$$\frac{}{\Xi;\Gamma \vdash (\lambda(x : U) \to v)\ u \overset{\beta}{\Rightarrow} v[{}^{u}/_{x}]} \tag{135}$$

PROJECTING A FIELD VALUE FROM A RECORD

$$\frac{}{\Xi;\Gamma \vdash \mathit{struct}\{\overrightarrow{f := u}\}.f \overset{\beta}{\Rightarrow} u} \quad [f := u \in \overrightarrow{f := u}] \tag{136}$$

Figure 46 Rules for β-reduction of functions and records

B.5.3 Eta conversion

Eta conversion (Figure 47) for functions and records describe something semantically equal, but where one instance may use redundant variables, for example for the sake of clarity, whereas the other is written in a more point-free manner. We define eta conversion as a non-directional conversion and not a reduction, although one could consider a natural normal form to be the most reduced one.

EXTRA ABSTRACTION WITH APPLICATION

$$\frac{\Xi;\Gamma \vdash t : (x : U) \to V}{\Xi;\Gamma \vdash t \overset{\eta}{=} \lambda(x : U) \to t\ x} \tag{137}$$

STRUCT OF PROJECTIONS

$$\frac{\Xi;\Gamma \vdash r : \mathit{sig}\{\overrightarrow{f : U}\}}{\Xi;\Gamma \vdash r \overset{\eta}{=} \mathit{struct}\{\overrightarrow{f := r.f}\}} \tag{138}$$

Figure 47 Rules for η-conversion of functions and records