Elaborating dependent (co)pattern matching

JESPER COCKX and ANDREAS ABEL, Department of Computer Science and Engineering, Chalmers and Gothenburg University, Sweden

In a dependently typed language, we can guarantee correctness of our programs by providing formal proofs. To check them, the typechecker elaborates these programs and proofs into a low level core language. However, this core language is by nature hard to understand by mere humans, so how can we know we proved the right thing? This question occurs in particular for dependent copattern matching, a powerful language construct for writing programs and proofs by dependent case analysis and mixed induction/coinduction. A definition by copattern matching consists of a list of clauses that are elaborated to a case tree, which can be further translated to primitive eliminators. In previous work this second step has received a lot of attention, but the first step has been mostly ignored so far.

We present an algorithm elaborating definitions by dependent copattern matching to a core language with inductive datatypes, coinductive record types, an identity type, and constants defined by well-typed case trees. To ensure correctness, we prove that elaboration preserves the first-match semantics of the user clauses. Based on this theoretical work, we reimplement the algorithm used by Agda to check left-hand sides of definitions by pattern matching. The new implementation is at the same time more general and less complex, and fixes a number of bugs and usability issues with the old version. Thus we take another step towards the formally verified implementation of a practical dependently typed language.

1 INTRODUCTION

Dependently typed functional languages such as Agda [2017], Coq [INRIA 2017], Idris [2013], and Lean [de Moura et al. 2015] combine programming and proving into one language, so they should be at the same time expressive enough to be useful and simple enough to be sound. These apparently contradictory requirements are addressed by having two languages: a high-level surface language that focuses on expressivity and a small core language that focuses on simplicity. The main role of the typechecker is then to elaborate the high-level surface language into the low-level core.

Since the difference between the surface and core languages can be quite large, the elaboration process can be, well, elaborate. If there is an error in the elaboration process, our program or proof may still be accepted by the system but its meaning is not what was intended [Pollack 1998]. In particular, the statement of a theorem may depend on the correct behaviour of some defined function, so if something went wrong in the elaboration of these definitions, the theorem statement may not be what it seems. As an extreme example, we may think we have proven an interesting theorem when in fact, we have only proven something trivial. This may be detected in a later phase when trying to use this proof, or it may not be detected at all. Unfortunately, there is no bulletproof way to avoid such problems: each part of the elaboration process has to be verified independently to make sure it produces something sensible.

One important part of the elaboration process is the elaboration of definitions by dependent pattern matching [Coquand 1992]. Dependent pattern matching provides a convenient high-level interface to the low-level constructions of case splitting, structural induction, and specialization by unification. The elaboration of dependent pattern matching goes in two steps: first the list of

\[1\] This paper is best viewed in color.

Authors’ address: Jesper Cockx, cockx@chalmers.se; Andreas Abel, andreas.abel@gu.se, Department of Computer Science and Engineering, Chalmers and Gothenburg University, Rännvägen 6b, Gothenburg, 41296, Sweden.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

clauses given by the user is translated to a case tree, and then the case tree is further translated to a term that only uses the primitive datatype eliminators. The second step has been studied in detail and is known to preserve the semantics of the case tree precisely \cite{Cockx2017,Goguen2006}. In contrast, the first step has received much less attention.

The goal of this paper is to formally describe an elaboration process of definitions by dependent pattern matching to a well-typed case tree for a realistic dependently typed language. Compared to the elaboration processes described by \citeauthor{Norell2007} \cite{Norell2007} and \citeauthor{Sozeau2010} \cite{Sozeau2010}, we make the following improvements:

- We include both pattern and copattern matching.
- We are more flexible in the placement of forced patterns.
- We prove that the translation preserves the first-match semantics of the user clauses.

We discuss each of these improvements in more detail below.

**Copatterns.** Copatterns provide a convenient way to define and reason about infinite structures such as streams \cite{Abel2013}. They can be nested and mixed with regular patterns. Elaboration of definitions by copattern matching has been studied for simply typed languages by \citeauthor{Setzer2014} \cite{Setzer2014}, but so far the combination of copatterns with general dependent types has not been studied in detail, even though it has already been implemented in Agda.

One complication when dealing with copatterns in a dependently typed language is that the type of a projection can depend on the values of the previous projections. For example, define the coinductive type \texttt{CoNat} of possibly infinite natural numbers by the two projections \texttt{iszero : Bool} and \texttt{pred : iszero \equiv Bool false \rightarrow CoNat}. We use copatterns to define the co-natural number \texttt{cozero}:

\begin{equation}
\begin{aligned}
\texttt{cozero : CoNat} \\
\texttt{cozero .iszero = true} \\
\texttt{cozero .pred \ emptyset}
\end{aligned}
\end{equation}

Here the new constant \texttt{cozero} is being defined with the field \texttt{iszero} equal to \texttt{true} (and no value for \texttt{pred}).

To refute the proof of \texttt{cozero .iszero \equiv Bool false} with an absurd pattern \texttt{emptyset}, the typechecker needs to know already that \texttt{cozero .iszero = true}, so it needs to check the clauses in the right order.

This example also shows that with mixed pattern/copattern matching, some clauses can have more arguments than others, so the typechecker has to deal with \textit{variable arity}. This means that we need to consider introducing a new argument as an explicit node in the constructed case tree.

**Flexible placement of forced patterns.** When giving a definition by dependent pattern matching that involves forced patterns (also called presupposed terms \cite{Brady2003} or inaccessible patterns \cite{Norell2007} or, in Agda, dot patterns), there are often multiple positions where to place them. For example, in the proof of symmetry of equality

\begin{equation}
\begin{aligned}
\texttt{sym : } (x : A) \rightarrow x \equiv_A y \rightarrow y \equiv_A x \\
\texttt{sym x [x] refl = refl}
\end{aligned}
\end{equation}

it should not matter if we instead write \texttt{sym [x] x refl = refl}. In fact, we even allow the apparently non-linear definition \texttt{sym x x refl = refl}. Our elaboration algorithm addresses this by treating forced patterns as \textit{laziness annotations}: they guarantee that the function will not match against a certain argument. This allows the user to be free in the placement of the forced patterns. For example, it is always allowed to write \texttt{zero} instead of \texttt{[zero]}, or \texttt{x} instead of \texttt{[x]}.

\footnote{In Agda, case trees are part of the core language so the second step is skipped in practice, but it is still important to know that it could be done in theory.}

With our elaboration algorithm, it is easy to extend the pattern syntax with forced constructor patterns such as \([\text{suc}]\ n\) (Brady et al. [2003]'s presupposed-constructor patterns). These allow the user to annotate that the function should not match on the argument but still bind some of the arguments of the constructor.

**Preservation of first-match semantics.** Like Augustsson [1985] and Norell [2007], we allow the clauses of a definition by pattern matching to overlap and use the first-match semantics in the construction of the case tree. For example, when constructing a case tree from the definition

\[
\begin{align*}
\text{max} : \mathbb{N} & \to \mathbb{N} \\
\text{max} \; \text{zero} \; y & = y \\
\text{max} \; x \; \text{zero} & = x \\
\text{max} \; (\text{suc} \; x) \; (\text{suc} \; y) & = \text{suc} \; (\text{max} \; x \; y)
\end{align*}
\]

we do not get \(\text{max} \; x \; \text{zero} = x\) but only \(\text{max} \; (\text{suc} \; x') \; \text{zero} = \text{suc} \; x'\). This makes a difference for dependent type checking where we evaluate open terms with free variables like \(x\). In this paper we provide a proof that the translation from a list of clauses to a case tree preserves the first-match semantics of the clauses. More precisely, we prove that if the arguments given to a function match a clause and all previous clauses produce a mismatch, then the case tree produced by elaborating the clauses also computes for the given arguments and the result is the same as the one given by the clause.

**Contributions.**

- We present a dependently typed core language with inductive datatypes, coinductive record types and an identity type. The language is focused [Andreoli 1992; Krishnaswami 2009; Zeilberger 2008]: terms of our language correspond to the non-invertible rules to introduce and eliminate these types, while the invertible rules constitute case trees.
- We are the first to present a coverage checking algorithm for fully dependent copatterns. Our algorithm desugars deep copattern matching to well-typed case trees in our core language.
- We prove correctness: if the desugaring succeeds, then the behaviour of the case tree corresponds precisely to the first-match semantics of the given clauses.
- We have implemented a new version of the algorithm used by Agda for checking the left-hand sides of a definition by dependent (co)pattern matching, which has been released as part of Agda 2.5.4. At the time of writing the effort to remodel the elaboration to a case tree according to the theory presented in this paper is still ongoing, but our work so far has already uncovered and fixed multiple issues in the old implementation [Agda issue 2017a,b,c,d, 2018a,b]. Our algorithm could also be used by other implementations of dependent pattern matching such as the Equations package for Coq [Sozeau 2010], Idris [2013], and Lean [de Moura et al. 2015].

This paper was born out of a practical need that arose while reimplementing the elaboration algorithm for Agda: it was not clear to us what exactly we wanted to implement, and we did not find sufficiently precise answers in the existing literature. Our main goal in this paper is therefore to give a precise description of the language, the elaboration algorithm, and the high-level properties we expect them to have. This also means we do not focus on fully developing the metatheory of the language or giving detailed proofs for all the basic properties one would expect.

3 Note that, in the example, the open term \(\text{max} \; x \; \text{zero}\) does not produce a mismatch with the first clause since it could match if variable \(x\) was replaced by \(\text{zero}\). In the first-match semantics, evaluation of \(\text{max} \; x \; \text{zero}\) is stuck.

4 Agda 2.5.4 released on 2018/06/02, changelog: https://hackage.haskell.org/package/Agda-2.5.4/changelog.
We start by introducing definitions by dependent (co)pattern matching and our elaboration algorithm to a case tree by a number of examples in Sect. 2. We then describe our core language in Sect. 3: the syntax, the rules for typing and equality, and the evaluation rules. In Sect. 4 we give the syntax and rules for case trees, and prove that a function defined by a well-typed case tree satisfies type preservation and coverage. Finally, in Sect. 5 we describe the rules for elaborating a definition by dependent (co)pattern matching to a well-typed case tree, and prove that this translation preserves the computational meaning of the given clauses. Sect. 6 discusses related work, and Sect. 7 concludes.

This is the short version of this article with most proofs omitted. A longer version with proofs can be found on the first author’s website.5

2 ELABORATING DEPENDENT (CO)PATTERN MATCHING BY EXAMPLE

Before we move on to the general description of our core language and the elaboration process, we give some examples of definitions by (co)pattern matching and how our algorithm elaborates them to a case tree. The elaboration works on a configuration \( \Gamma \vdash P \mid u : C \) consisting of:

- A context \( \Gamma \), i.e. a list of variables annotated with types. Initially \( \Gamma \) is the empty context \( \epsilon \).
- The current target type \( C \). This type may depend on variables bound in \( \Gamma \). Initially \( C \) is the type of the function being defined.
- A representation of the left-hand side \( u \). In the end \( u \) should have type \( C \) in context \( \Gamma \). Initially \( u \) is the function being defined itself.
- A list of partially deconstructed user clauses \( P \). Initially these are the clauses as written by the user.

These four pieces of data together describe the current state of elaborating the definition.

The elaboration algorithm transforms this state step by step until the user clauses are deconstructed completely. In the examples below, we annotate each step with a label such as SPLITCON or INTRO, linking it to the general rules given in Sect. 5.

Example 1. Let us define a function \( \text{max} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \) by pattern matching as in the introduction (3). The initial configuration is \( \Gamma \vdash \max : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \) where

\[
P_0 = \begin{cases} 
\text{zero} \quad j & \mapsto j \\
i \quad \text{zero} & \mapsto i \\
(suc \ k) \quad (suc \ l) & \mapsto suc (\max \ k \ l)
\end{cases}
\]

The first operation we need is to introduce a new variable \( m \) (rule INTRO). It transforms the initial problem into \( (m : \mathbb{N}) \vdash P_1 \mid \max : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \) where

\[
P_1 = \begin{cases} 
[m \mapsto \text{zero}] \quad j & \mapsto j \\
[m \mapsto i] \quad \text{zero} & \mapsto i \\
[m \mapsto suc \ k] \quad (suc \ l) & \mapsto suc (\max \ k \ l)
\end{cases}
\]

This operation strips the first user pattern from each clause and replaces it by a constraint \( m \mapsto p \) that it should be equal to the newly introduced variable \( m \). We write these constraints between brackets in front of each individual clause.

The next operation we need is to perform a case analysis on the variable \( m \) (rule \textit{SplitCon}).\footnote{At this point we could also introduce the variable for the second argument of \textit{max}, the elaboration algorithm is free to choose either option.} This transforms the problem into two subproblems \( \Gamma \vdash P_2 \mid \text{max} \ zero : \mathbb{N} \rightarrow \mathbb{N} \) and \( \Gamma \vdash P_3 \mid \text{max} (\text{suc} \ p) : \mathbb{N} \rightarrow \mathbb{N} \) where

\[
P_2 = \begin{cases} 
\text{zero} \not\rightarrow j & \text{zero} \not\rightarrow i \\
\text{suc} \ p \ not \ suc \ k & \text{suc} \ (\text{max} \ k \ l)
\end{cases} \quad (6)
\]

\[
P_3 = \begin{cases} 
\text{zero} \not\rightarrow j & \text{zero} \not\rightarrow i \\
\text{suc} \ p \ not \ suc \ k & \text{suc} \ (\text{max} \ k \ l)
\end{cases} \quad (7)
\]

We simplify the constraints, removing those clauses with absurd constraints:

\[
P_2 = \begin{cases} 
\text{zero} \not\rightarrow j & \text{zero} \not\rightarrow i \\
\text{suc} \ p \ not \ suc \ k & \text{suc} \ (\text{max} \ k \ l)
\end{cases} \quad (8)
\]

We continue applying these operations \textit{Intro} and \textit{SplitCon} (introducing a new variable and case analysis on a variable) until the first clause has no more user patterns and no more constraints where the left-hand side is a constructor. For example, for \( P_2 \) we get after one more introduction step \( (n : \mathbb{N}) \vdash P_4 \mid \text{max} \ zero \ n : \mathbb{N} \) where

\[
P_4 = \begin{cases} 
[\text{zero} \not\rightarrow j] & [\text{zero} \not\rightarrow i] \\
[\text{zero} \not\rightarrow i, n \not\rightarrow \text{zero}] & [\text{zero} \not\rightarrow i]
\end{cases} \quad (9)
\]

We solve the remaining constraint in the first clause by instantiating \( j := n \). This means we are done and we have \text{max} \ zero \ n = f(n / j) = n \ (\text{rule \ 	extit{Done}}). \ Similarly, elaborating \( (p : \mathbb{N}) \vdash P_3 \mid \text{max} (\text{suc} \ p) : \mathbb{N} \rightarrow \mathbb{N} \) (with rules \textit{Intro}, \textit{SplitCon}, and \textit{Done}) gives us \text{max} (\text{suc} \ p) \zero = \text{suc} \ p \text{ and max} (\text{suc} \ p) \ (\text{true} \ q) = \text{suc} (\text{max} \ p \ q).

We record the operations used when elaborating the clauses in a \textit{case tree}. Our syntax for case trees is close to the normal term syntax in other languages: \( \lambda x . \) for introducing a new variable and \text{case}_{x}[] \) for a case split. For \textit{max}, we get the following case tree:

\[
\lambda m. \text{case}_{m} \begin{cases} 
\text{zero} & \rightarrow \lambda n. \ n \\
\text{suc} \ p & \rightarrow \lambda n. \ 	ext{case}_{n} \begin{cases} 
\text{zero} & \rightarrow \text{suc} \ p \\
\text{suc} \ q & \rightarrow \text{suc} (\text{max} \ p \ q)
\end{cases}
\end{cases}
\]

\quad(10)

\textbf{Example 2.} Next we take a look at how to elaborate definitions using copatterns. For the \textit{cozero} example (1), we have the initial configuration \( \Gamma \vdash P_{0} \mid \text{cozero} : \text{CoNat} \) where:

\[
P_{0} = \begin{cases} 
\text{.iszero} & \rightarrow \text{true} \\
\text{.pred} & \rightarrow \text{impossible}
\end{cases} \quad (11)
\]

Here we need a new operation to split on the result type \textit{CoNat} (rule \textit{Cosplit}). This produces two subproblems \( \Gamma \vdash P_{1} \mid \text{cozero} \ .\text{iszero} \) and \( \Gamma \vdash P_{2} \mid \text{cozero} \ .\text{pred} : \text{cozero} \ .\text{iszero} \equiv_{\text{Bool}} \text{false} \rightarrow \text{CoNat} \) where

\[
P_{1} = \{ \rightarrow \text{true} \} \quad P_{2} = \emptyset \rightarrow \text{impossible} \quad (12)
\]

The first problem is solved immediately with \textit{cozero} \ .\text{iszero} = \text{true} \ (\text{rule \ 	extit{Done}}). \ In the second problem we introduce the variable \( x : \text{cozero} \ .\text{iszero} \equiv_{\text{Bool}} \text{false} \ (\text{rule \ 	extit{Intro}}) \) and note that \textit{cozero} \ .\text{iszero} = \text{true} \ from \ the \ previous \ branch, \ hence \( x : \text{true} \equiv_{\text{Bool}} \text{false}. \ Since \text{true} \equiv_{\text{Bool}} \text{false} \ is \ an \ empty \ type \ (\text{technically, since unification of \textit{true} with \textit{false} results in a conflict}), \ we \ can \ perform \ a \ case \ split \ on \ \( x \) \! with \ zero \ cases \ (\text{rule \ 	extit{SplitEmpty}}, \ solving \ the \ problem.\)
In the resulting case tree, the syntax for a split on the result type is \texttt{record\{}\}, with one subtree for each field of the record type:
\[
\texttt{record}\left\{ \begin{array}{l}
iszero \mapsto \text{true} \\
pred \mapsto \lambda \mathbf{x} . \text{case}_\mathbf{x}\{ \}
\end{array}\right. \quad \text{(13)}
\]
For the next examples, we omit the details of the elaboration process and only show the definition by pattern matching and the resulting case tree.

\textbf{Example 3.} Consider the type \texttt{CStream} of C streams: potentially infinite streams of numbers that end on a zero. We define this as a record where the \texttt{tail} field has two extra arguments enforcing that we can only take the tail if the head is \texttt{suc \ m} for some \texttt{m}.
\[
\texttt{record self : CStream : Set where}
\begin{align*}
\text{head} &: \mathbb{N} \\
\text{tail} &: (m : \mathbb{N}) \rightarrow \text{self} . \text{head} \equiv_\mathbb{N} \text{suc \ m} \rightarrow \text{CStream}
\end{align*}
\quad \text{(14)}
\]
Here, the name \texttt{self} is bound to the current record instance, allowing later projections to depend on prior projections.

Now consider the function \texttt{countdown} that creates a C stream counting down from a given number \texttt{n}:
\[
\begin{align*}
\text{countdown} &: \mathbb{N} \rightarrow \text{CStream} \\
\text{countdown \ n} . \text{head} &= n \\
\text{countdown \ zero} . \text{tail} &= m \ \emptyset \\
\text{countdown (suc \ m)} . \text{tail} &= m \ \text{refl} = \text{countdown \ m}
\end{align*}
\quad \text{(15)}
\]
Our elaboration algorithm applies rules \texttt{Intro}, \texttt{Cosplit}, \texttt{SplitCon}, \texttt{SplitEmpty}, \texttt{SplitEq}, and \texttt{Done} in sequence to translate this definition to the following case tree:
\[
\lambda n . \text{record}\left\{ \begin{array}{l}
\text{head} \mapsto n \\
\text{tail} \mapsto \lambda m, p . \text{case}_n \left\{ \begin{array}{l}
\text{zero} \mapsto \text{case}_p\{ \}
\text{suc \ n'} \mapsto \text{case}_p \left\{ \text{refl} \mapsto^1_m (\text{countdown \ m}) \right\} \end{array}\right. \}
\end{array}\right. \quad \text{(16)}
\]
Note the extra annotation \texttt{\_ m} after the case split on \texttt{p : suc \ m} \equiv_\mathbb{N} \text{suc \ n}'. This is a substitution (in this case the identity substitution on \texttt{(m : \mathbb{N})}) necessary for the evaluation rules of the case tree when matching on \texttt{refl}. It reflects the fact that \texttt{n'} went out of scope after the case split on \texttt{refl} : \texttt{suc \ n' \equiv_\mathbb{N} suc \ m} (since unification instantiated it with \texttt{m}) so only the variable \texttt{m} can still be used after this point.

\textbf{Example 4.} This example is based on issue \#2896 on the Agda bug tracker [Agda issue 2018b].
The problem was that Agda’s old elaboration algorithm threw away a part of the pattern written by the user. This meant the definition could be elaborated to a different case tree from the one intended by the user.

The (simplified) example consists of the following datatype \texttt{D} and function \texttt{foo}:
\[
\begin{align*}
\text{data} \ D \ (m : \mathbb{N}) &: \text{Set where} \\
\text{c} &: (n : \mathbb{N}) \ (p : n \equiv_\mathbb{N} m) \rightarrow D \ m \\
\text{foo} &: (m : \mathbb{N}) \rightarrow D \ (\text{suc \ m}) \rightarrow \mathbb{N} \\
\text{foo} \ m \ (c \ (\text{suc \ n}) \ \text{refl}) &= m + n
\end{align*}
\quad \text{(17)}
\]
The old algorithm would ignore the pattern \texttt{suc \ n} in the definition of \texttt{foo} because it corresponds to a forced pattern after the case split on \texttt{refl}. Our elaboration instead produces the following case tree (using rules \texttt{Intro}, \texttt{SplitCon}, \texttt{SplitEq}, and \texttt{Done}):  
\[
\lambda m, x . \text{case}_x\left\{ \begin{array}{l}
c \ n \ p \mapsto \text{case}_p \left\{ \text{refl} \mapsto^1_m (m + m) \right\} \end{array}\right. \quad \text{(18)}
\]
Elaborating dependent (co)pattern matching

Even though this case tree does not match on the \texttt{suc} constructor, it implements the same computational behaviour as the clause in the definition of \texttt{foo} because the first argument of \texttt{c} is \texttt{forced} to be \texttt{suc m} by the typing rules.

This example also shows another feature supported by our elaboration algorithm, namely that two different variables \texttt{m} and \texttt{n} in the user syntax may correspond to the same variable \texttt{m} in the core syntax. In effect, \texttt{n} is treated as a let-bound variable with value \texttt{m}.

Example 5. This example is based on issue #2964 on the Agda bug tracker [Agda issue 2018a]. The problem was that Agda was using a too liberal version of the first-match semantics that was not preserved by the translation to a case tree. The problem occurred for the following definition:

\[
\begin{align*}
\textit{f} : & (A : \textit{Set}) \rightarrow A \rightarrow \textit{Bool} \rightarrow (A \equiv_{\textit{Set}} \textit{Bool}) \rightarrow \textit{Bool} \\
\textit{f} & \{\textit{Bool}\} \textit{true true refl} = \textit{true} \\
\textit{f} & \_ \_ \_ = \textit{false}
\end{align*}
\]

This function is elaborated (both by Agda’s old algorithm and by ours) to the following case tree (using rules \texttt{Intro}, \texttt{SplitCon}, \texttt{SplitEq}, and \texttt{Done}):

\[
\begin{align*}
\lambda A, x, y, p. \texttt{case}_y \left\{ \begin{array}{ll}
\texttt{true} & \rightarrow \texttt{case}_p \left\{ \begin{array}{ll}
\texttt{refl} & \rightarrow \_x, y \\
\texttt{case}_x & \left\{ \begin{array}{ll}
\texttt{true} & \rightarrow \texttt{true} \\
\texttt{false} & \rightarrow \texttt{false}
\end{array} \right. \\
\texttt{false} & \rightarrow \texttt{false}
\end{array} \right.
\end{array} \right.
\end{align*}
\]

According to the (liberal) first-match semantics, we should have \textit{f Bool false y p} = \textit{false} for any \textit{y : Bool} and \textit{p : Bool} \equiv_{\textit{Set}} \textit{Bool}, but this is not true for the case tree since evaluation gets stuck on the variable \textit{y}. Another possibility is to start the case tree by a split on \textit{p} (after introducing all the variables), but this case tree still gets stuck on the variable \textit{p}. In fact, there is no well-typed case tree that implements the first-match semantics of these clauses since we cannot perform a case split on \textit{x : A} before splitting on \textit{p}.

One radical solution for this problem would be to only allow case trees where the case splits are performed in order from left to right. However, this would mean the typechecker must reject many definitions such as \textit{f} in this example, because the type of \textit{x} is not known to be a datatype until the case split on \textit{A} \equiv_{\textit{Set}} \textit{Bool}. Instead we choose to keep the elaboration as it is and restrict the first-match semantics of clauses. In the example of \textit{f}, this change means that we can only go to the second clause once all three arguments \textit{x}, \textit{y} and \textit{p} are constructors, and at least one of them produces a mismatch.

3 CORE LANGUAGE

In this section we introduce a basic type theory for studying definitions by dependent (co)pattern matching. It has support for dependent function types, an infinite hierarchy of predicative universes, equality types, inductive datatypes and coinductive records.

To keep the work in this paper as simple as possible, we leave out many features commonly included in dependently typed languages, such as lambda expressions and inductive families of datatypes (other than the equality type). These features can nevertheless be encoded in our language, see Sect. 3.5 for details.

Note also that we do not include any rules for \(\eta\)-equality, neither for lambda expressions (which do not exist) nor for records (which can be coinductive hence do not satisfy \(\eta\)). Sect. 3.5 discusses how our language could be extended with \(\eta\)-rules.
3.1 Syntax of the core type theory

Expressions of our type theory are almost identical to Agda’s internal term language. All function applications are in spine-normal form, so the head symbol of an application is exposed, be it variable \( x \), data \( D \) or record type \( R \), or defined function \( f \). We generalize applications to eliminations \( e \) by including projections \( \pi \) in spines \( \tilde{e} \). Any expression is in weak head normal form but \( f \tilde{e} \), which is computed via pattern matching (see Sect. 3.4).

\[
\begin{align*}
A, B, u, v & ::= w & \text{weak head normal form} \\
W, w & ::= (x : A) \rightarrow B & \text{dependent function type} \\
| \text{Set}_\ell & & \text{universe } \ell \\
| D \tilde{u} & & \text{datatype fully applied to parameters} \\
| R \tilde{u} & & \text{record type fully applied to parameters} \\
| u & \equiv_A v & \text{equality type} \\
| x \bar{e} & & \text{variable applied to eliminations} \\
| c \tilde{u} & & \text{constructor fully applied to arguments} \\
| \text{refl} & & \text{proof of reflexivity} \\
\end{align*}
\]

Any expression but \( c \tilde{u} \) or \( \text{refl} \) can be a type; the first five weak head normal forms are definitely types. Any type has in turn a type, specifically some universe \( \text{Set}_\ell \). Syntax is colored according to the Agda conventions: primitives and defined symbols are blue, constructors are green, and projections are pink.

\[
\begin{align*}
e & ::= u & \text{application} \\
| \cdot \pi & & \text{projection}
\end{align*}
\]

Binary application \( u \ e \) is defined as a partial function on the syntax: for variables and functions it is defined by \( (x \bar{e}) \ e = x (\bar{e} ; e) \) and \( (f \bar{e}) \ e = f (\bar{e} ; e) \) respectively, otherwise it is undefined.

Patterns are generated from variables and constructors. In addition, we have forced and absurd patterns. Since we are matching spines, we also consider projections as patterns, or more precisely, as copatterns.

\[
\begin{align*}
p & ::= x & \text{variable pattern} \\
| \text{refl} & & \text{pattern for reflexivity proof} \\
| c \bar{p} & & \text{constructor pattern} \\
| [c] \bar{p} & & \text{forced constructor pattern} \\
| [u] \bar{p} & & \text{forced argument} \\
| \emptyset & & \text{absurd pattern}
\end{align*}
\]

Forced patterns [Brady et al. 2003] appear with dependent types; they are either entirely forced arguments \( [u] \), which are Agda’s dot patterns, or only the constructor is forced \([c] \bar{p}\). An argument can be forced by a match against \( \text{refl} \) somewhere in the surrounding (co)pattern. However, sometimes we want to bind variables in a forced argument; in this case, we revert to forced constructors. Absurd patterns\(^7\) are used to indicate that the type at this place is empty, i.e. no constructor can possibly match. They are also used to indicate an empty copattern split, i.e. a copattern split on a

---

\(^7\)Absurd patterns are written () in Agda syntax.
record type with no projections. This allows us in particular to define the unique element tt of the
unit record, which has no projections at all, by the clause \( \text{tt} \; \emptyset = \text{impossible} \).

The pattern variables \( \text{PV}(q) \) is the list of variables in \( q \) that appear outside forcing brackets \( \lfloor \cdot \rfloor \). By
removing the forcing brackets, patterns \( p \) embed into terms \( \lfloor p \rfloor \), and copatterns \( q \) into eliminations
\( \lfloor q \rfloor \), except for the absurd pattern \( \emptyset \).

\[
\begin{align*}
[x] &= x \\
[\text{refl}] &= \text{refl} \\
[\lfloor c \; \rho \rfloor] &= c \lfloor \rho \rfloor \\
[\lfloor u \rfloor] &= u
\end{align*}
\]  
\( (24) \)

Constructors take a list of arguments whose types can depend on all previous arguments. The
constructor parameters are given as a list \( x_1:A_1, \ldots, x_n:A_n \) with pairwise distinct \( x_i \) where \( A_i \) can
depend on \( x_1, \ldots, x_{i-1} \). This list can be conceived as a cons-list, then it is called a telescope, or as a
snoc-list, then we call it a context.

\[
\begin{align*}
\Gamma ::= & \; \epsilon \quad \text{empty context} & \Delta ::= & \; \epsilon \quad \text{empty telescope} \\
| & \; \Gamma(x:A) \Delta \quad \text{context extension} & | & \; (x:A)\Delta \quad \text{non-empty telescope}
\end{align*}
\]  
\( (25) \)

Context and telescopes can be regarded as finite maps from variables to types, and we require
\( x \not\in \text{dom}(\Gamma) \) and \( x \not\in \text{dom}(\Delta) \) in the above grammars. We implicitly convert between contexts
and telescopes, but there are still some conceptual differences. Contexts are always closed, i.e. its
types only refer to variables bound prior in the same context. In contrast, we allow open telescopes
whose types can also refer to some surrounding context. Telescopes can be naturally thought of as
context extensions, and if \( \Gamma \) is a context and \( \Delta \) a telescope in context \( \Gamma \) where \( \text{dom}(\Gamma) \) and \( \text{dom}(\Delta) \)
are disjoint, then \( \Gamma\Delta \) defined by \( \Gamma\epsilon = \Gamma \) and \( \Gamma((x:A)\Delta) = (\Gamma(x:A))\Delta \) is a new valid context. We
embed telescopes in the syntax of declarations, but contexts are used in typing rules exclusively.

Given a telescope \( \Delta \), let \( \overline{\Delta} \) be \( \Delta \) without the types, i.e. the variables of \( \Delta \) in order. Further, we
define \( \overline{\Delta \to C} \) as the iterated dependent function type via \( \epsilon \to C = C \) and \( (x:A)\overline{\Delta \to C} = (x:A) \to \overline{\Delta \to C} \).

A development in our core type theory is a list of declarations, of which there are three kinds:
data type, record type, and function declarations. The input to the type checker is a list of unchecked
declarations \( \text{decl}^\circ \), and the output a list of checked declarations \( \text{decl}^\circ \), called a signature \( \Sigma \).

\[
\begin{align*}
s ::= & \; \emptyset \quad \text{status: unchecked} & | & \; \emptyset \quad \text{status: checked} \\
\text{decl}^\circ ::= & \; \text{data} \; D : A \; \text{where} \; \text{con} \quad \text{datatype declaration} \\
| & \; \text{record} \; self : R : \Delta \; \text{where} \; \text{field} \quad \text{record declaration} \\
| & \; \text{definition} \; f : A \; \text{where} \; \text{cls}^\circ \quad \text{function declaration} \\
\text{con} ::= & \; c \; \Delta \quad \text{constructor declaration} \\
\text{field} ::= & \; \pi : A \quad \text{field declaration} \\
\text{cls}^\circ ::= & \; \overline{\rho} \to \text{rhs} \quad \text{unchecked clause} \\
\text{cls}^\circ ::= & \; \Delta \vdash \overline{\rho} \to \text{u} : B \quad \text{checked clause} \\
\text{rhs} ::= & \; \text{u} \quad \text{clause body: expression} \\
| & \; \text{impossible} \quad \text{empty body for absurd pattern} \\
\Sigma ::= & \; \text{decl}^\circ \quad \text{signature}
\end{align*}
\]  
\( (26) \)
A *data type* \( D \) can be parameterized by telescope \( \Delta \) and inhabits one of the universes \( \text{Set}_\ell \). Each of its constructors \( c_i \) (although there might be none) takes a telescope \( \Delta_i \) of arguments that can refer to the parameters in \( \Delta \). The full type of \( c_i \) could be \( \Delta \Delta_i \to D \bar{\Delta} \), but we never apply constructors to the data parameters explicitly.

A *record type* \( R \) can be thought of as a single constructor data type; its fields \( \pi_1 : A_1, \ldots, \pi_n : A_n \) would be the constructor arguments. The field list behaves similar to a telescope, the type of each field can depend on the value of the previous fields. However, these values are referred to via \( \text{self} . \pi_i \) where variable \( \text{self} \) is a placeholder for the value of the whole record.\(^8\) The full type of projection \( \pi_i \) could be \( \Delta(\text{self} : R \bar{\Delta}) \to A_i \), but like for constructors, we do not apply a projection explicitly to the record parameters.

Even though we do not spell out the conditions for ensuring totality in this paper, like *positivity, termination*, and *productivity* checking, data types, when recursive, should be thought of as inductive types, and record types, when recursive, as coinductive types [Abel et al. 2013]. Thus, there is no dedicated constructor for records; instead, concrete records are defined by what their projections compute.

Such definitions are subsumed under the last alternative dubbed *function declaration*. More precisely, these are *definitions by copattern matching* which include record definitions. Each clause defining the constant \( f : A \) consists of a list of copatterns \( q \) and right hand side \( \text{rhs} \). The copatterns eliminate type \( A \) into the type of the \( \text{rhs} \) which is either a term \( u \) or the special keyword \( \text{impossible} \), in case one of the copatterns \( q_i \) contains an absurd pattern \( \emptyset \). The intended semantics is that if an application \( f e \) matches a left hand side \( f q \) with substitution \( \sigma \), then \( f e \) reduces to \( \text{rhs} \) under \( \sigma \). For efficient computation of matching, we require *linearity of pattern variables* for checked clauses: each variable in \( q \) occurs only once in a non-forced position.

While checking declarations, the typechecker builds up a signature \( \Sigma \) of already checked (parts of) declarations. Checked clauses are the elaborations (sections 2 and 5) of the corresponding unchecked clauses: they are non-overlapping and supplemented by a telescope \( \Delta \) holding the types of the pattern variables and the type \( B \) of left and right hand side. Further, checked clauses do not contain absurd patterns.

In the signature, the last entry might be incomplete, e.g. a data type missing some constructors, a record type missing some fields, or a function missing some clauses. During checking a declaration, we might add already checked parts of the declaration, dubbed *snippets*, to the signature.

\[
Z ::= \begin{array}{l}
data \text{ type signature} \\
\mid \text{ constructor } c \Delta_c : D \Delta \\
\mid \text{ record } R \Delta : \text{Set}_\ell \\
\mid \text{ projection } \text{self} : R \Delta \vdash . \pi : A \\
\mid \text{ definition } f : A \\
\mid \text{ clause } \Delta \vdash f \bar{q} \leftrightarrow u : B \\
\end{array}
\]

(27)

Adding a snippet \( Z \) to a signature \( \Sigma \), written \( \Sigma[Z] \) is always defined if \( Z \) is a data or record type or function signature; in this case, the corresponding declaration is appended to \( \Sigma \). Adding a constructor signature \( \text{constructor } c \Delta_c : D \Delta \) is only defined if the last declaration in \( \Sigma \) is \( \text{(data } D \Delta : \text{Set}_\ell \text{ where } \overline{\text{cof}}) \) and \( c \) is not part of \( \overline{\text{cof}} \) yet. Analogous conditions apply when adding projection snippets. Function clauses can be added if the last declaration of \( \Sigma \) is a function declaration with the same name. We trust the formal definition of \( \Sigma[Z] \) to the imagination of the reader. The conditions ensure that we do not add new constructors to a data type that is already complete or new fields to a completed record declaration. Such additions could destroy coverage.

\(^8\) \text{self is the analogous of Java’s this, but like in Scala’s } \text{trait}, \text{ the name can be chosen.}
for functions that have already been checked. Late addition of function clauses would not pose a problem, but that feature would be obsolete for our type theory anyway.

Membership of a snippet is written $Z \in \Sigma$ and a decidable property with the obvious definition. These operations on the signature will be used in the inference rules of our type theory. Since we only refer to a constructor $c$ in conjunction with its data type $D$, constructors can be overloaded, and likewise projections.

3.2 Typing and equality

Our type theory employs the following basic typing and equality judgments, which are relative to a signature $\Sigma$.

$$
\Sigma \vdash \Gamma \quad \text{context } \Gamma \text{ is well-formed}
$$

$$
\Sigma; \Gamma \vdash \Delta \quad \text{in context } \Gamma, \text{ telescope } \Delta \text{ is well-formed and } \ell\text{-bounded}
$$

$$
\Sigma; \Gamma \vdash \bar{u} : \Delta \quad \text{in context } \Gamma, \text{ term list } \bar{u} \text{ instantiates telescope } \Delta
$$

$$
\Sigma; \Gamma \vdash \varnothing : A \quad \text{in context } \Gamma, \text{ head } u \text{ of type } A \text{ is eliminated via } \bar{e} \text{ to type } B
$$

$$
\Sigma; \Gamma \vdash u = \nu : A \quad \text{in context } \Gamma, \text{ terms } u \text{ and } \nu \text{ are equal of type } A
$$

$$
\Sigma; \Gamma \vdash \bar{u} = \bar{\nu} : \Delta \quad \text{in context } \Gamma, \text{ term lists } \bar{u} \text{ and } \bar{\nu} \text{ are equal instantiations of } \Delta
$$

$$
\Sigma; \Gamma \vdash A + \bar{e} = \bar{e} : B \quad \bar{e} \text{ and } \bar{e}' \text{ are equal eliminations of head } u : A \text{ to type } B \text{ in } \Gamma
$$

In all these judgements, the signature $\Sigma$ is fixed, thus we usually omit it, e.g. in the inferences rules.

We further define some shorthands for type-level judgements when we do not care about the universe level $\ell$:

$$
\Sigma; \Gamma \vdash \Delta \quad \iff \quad \exists \ell. \quad \Sigma; \Gamma \vdash_{\ell} \Delta \quad \text{well-formed telescope}
$$

$$
\Sigma; \Gamma \vdash A \quad \iff \quad \exists \ell. \quad \Sigma; \Gamma \vdash_{\ell} A : \text{Set}_{\ell} \quad \text{well-formed type}
$$

$$
\Sigma; \Gamma \vdash A = B \quad \iff \quad \exists \ell. \quad \Sigma; \Gamma \vdash_{\ell} A = B : \text{Set}_{\ell} \quad \text{equal types}
$$

In the inference rules, we make use of substitutions. Substitutions $\sigma, \tau, \nu$ are partial maps from variable names to terms with a finite domain. If $\text{dom}(\sigma)$ and $\text{dom}(\tau)$ are disjoint, then $[\sigma \cup \tau]$ denotes the union of these maps. We write the substitution that maps the variables $x_1, \ldots, x_n$ to the terms $v_1, \ldots, v_n$ (and is undefined for all other variables) by $[v_1 / x_1; \ldots; v_n / x_n]$. In particular, the empty substitution $[]$ is undefined for all variables. If $\Delta = (x_1 : A_1) \ldots (x_n : A_n)$ is a telescope and $\varnothing = v_1, \ldots, v_n$ is a list of terms, we may write $[\varnothing / \Delta]$ for the substitution $[\varnothing / \hat{\Delta}]$, i.e. $[v_1 / x_1; \ldots; v_n / x_n]$. In particular, the identity substitution $1_{\Gamma} = [\hat{\Gamma} / \Gamma]$ maps all variables in $\Gamma$ to themselves. We also use the identity substitution as a weakening substitution, allowing us to forget about all variables that are not in $\Gamma$. If $x \in \text{dom}(\sigma)$, then $[\sigma \setminus x]$ is defined by removing $x$ from the domain of $\sigma$.

Application of a substitution $\sigma$ to a term $u$ is written as $[u]_{\sigma}$ and is defined as usual by replacing all (free) variables in $u$ by their values given by $\sigma$, avoiding variable capture via suitable renaming of bound variables. Like function application, this is a partial operation on the syntax; for instance, $(x \cdot \pi)[c / x]$ is undefined as constructors cannot be the head of an elimination. Thus, when a substitution appears in an inference rule, its definedness is an implicit premise of the rule. Also, such pathological cases are ruled out by typing. Well-typed substitutions can always be applied to well-typed terms. Substitution composition $[\sigma ; \tau]$ shall map the variable $x$ to the term $(x \sigma) \tau$. Note the difference between $\sigma ; \tau$ and $\sigma \cup \tau$: the former applies first $\sigma$ and then $\tau$ in sequence, while the latter applies $\sigma$ and $\tau$ in parallel to disjoint parts of the context. Application of a substitution to a pattern $[p]_{\sigma}$ is defined as $[p]_{\sigma}$.

In addition to substitutions on terms, we also make use of substitutions on patterns called pattern substitutions. A pattern substitution $\rho$ assigns to each variable a pattern. We reuse the same syntax
Values.

\[ \Gamma \vdash u : A \] Entails \( \Gamma \) and \( \Gamma \vdash A \).

Types.

\[ \vdash \Gamma \quad \vdash \Gamma \vdash A : \text{Set}_\ell \quad \Gamma \vdash (x : A) \vdash B : \text{Set}_{\ell'} \]

\[ \Gamma \vdash \text{Set}_\ell : \text{Set}_{\ell+1} \]

\[ \Gamma \vdash (x : A) \rightarrow B : \text{Set}_{\max(\ell, \ell')} \]

\[ \Gamma \vdash \text{data} D \Delta : \text{Set}_\ell \subseteq \Sigma \]

\[ \Gamma \vdash \text{record} R \Delta : \text{Set}_\ell \subseteq \Sigma \]

\[ \Gamma \vdash \bar{u} : \Delta \]

\[ \Gamma \vdash R \bar{u} : \text{Set}_\ell \]

\[ \Gamma \vdash A : \text{Set}_\ell \quad \Gamma \vdash u : A \quad \Gamma \vdash v : A \]

\[ \Gamma \vdash u \equiv_A v : \text{Set}_\ell \]

Heads \( h := x \in f.e \) and applications \( h \bar{e} \).

\[ \vdash \Gamma \quad \Gamma \vdash x : A \in \Gamma \]

\[ \Gamma \vdash x \in e : A \quad \Gamma \vdash f \in e : A \]

\[ \Gamma \vdash h : A \quad \Gamma \vdash h \in e : C \]

Values.

\[ \frac{\text{constructor} \; c \Delta_c : D \Delta \subseteq \Sigma \quad \Gamma \vdash \bar{u} : \Delta \quad \Gamma \vdash \bar{v} : \Delta_c[\bar{u} / \Delta]}{\Gamma \vdash c \bar{v} : D \bar{u}} \]

\[ \Gamma \vdash u : A \]

\[ \Gamma \vdash \text{refl} : u \equiv_A u \]

Conversion.

\[ \Gamma \vdash u : A \quad \Gamma \vdash A = B \]

\[ \Gamma \vdash u : B \]

Fig. 1. Typing rules for expressions.

\[ \Gamma \vdash u : A \vdash e : C \] If \( \Gamma \vdash u : A \) then \( \Gamma \vdash C \).

\[ \frac{\Gamma \vdash v : A \quad \Gamma \vdash \bar{u} : B[\bar{v} / x] \vdash e : C}{\Gamma \vdash \bar{u} : A \rightarrow B} \]

\[ \frac{\Gamma \vdash \bar{v} \vdash .\pi \bar{e} : C}{\Gamma \vdash \text{projection} \; \text{self} : R \Delta \vdash .\pi : A \subseteq \Sigma \quad \Gamma \vdash u \quad .\pi : A[\bar{v} / \Delta, u / \text{self}] \vdash e : C} \]

\[ \Gamma \vdash u : R \bar{v} \vdash .\pi \bar{e} : C \]

\[ \Gamma \vdash A = A' \quad \Gamma \vdash u : A' + e : C \]

\[ \Gamma \vdash u : A + e : C \]

Fig. 2. The typing rules for eliminations.

for pattern substitutions as for normal substitutions. Any pattern substitution \( \rho \) can be used as a normal substitution \( \sigma \rho \) defined by \( x[\rho \sigma \rho] = [x\rho] \).

The rules for the typing judgement \( \Gamma \vdash t : A \) are listed in Fig. 1. The type formation rules introduce an infinite hierarchy of predicative universes \( \text{Set}_\ell \) without cumulativity. The formation rules for data and record types make use of the judgment \( \Gamma \vdash u : \Delta \) to type argument lists, same for the constructor rule, which introduces a data type. Further, \( \text{refl} \) introduces the equality type.
with the non-invertible rules, the terms, while the choice-free pattern matching corresponding to
when checking that the level of a data type is sufficiently high for the level of data it contains
All expressions involved in these rules are fully applied, but this changes when we come to the
elimination rules. The types of heads, i.e. variables x or defined constants f are found in the context
or signature.

The rules for applying heads u to spines ē, judgement \( \Gamma \vdash u : A \vdash ē : C \), are presented in Fig. 2. For checking arguments, the type of the head is sufficient, and it needs to be a function type. To check projections, we need also the value u of the head that replaces self in the type of the projection. We may need to convert the type of the head to a function or record type to apply these rules, hence, we supply a suitable conversion rule. The result type C of this judgement need not be converted here, it can be converted in the typing judgement for expressions.

Remark 6 (Focused syntax). The reader may have observed that our expressions cover only the non-invertible rules in the sense of focusing [Andreoli 1992], given that we consider data types as multiplicative disjunctions and record types as additive conjunctions: Terms introduce data and eliminate records and functions. The invertible rules, i.e. elimination for data and equality and introduction for function space and records are covered by pattern matching (Sect. 3.4) and, equivalently, case trees (Sect. 4). This matches our intuition that all the information/choice resides with the non-invertible rules, the terms, while the choice-free pattern matching corresponding to the invertible rules only sets the stage for the decisions taken in the terms.

Fig. 3 defines judgement \( \Gamma \vdash \ell : \Delta \) for telescope formation. The level \( \ell \) is an upper bound for the universe levels of the types that comprise the telescope. In particular, if we consider a telescope as a nested \( \Sigma \)-type, then \( \ell \) is an upper bound for the universe that hosts this type. This is important when checking that the level of a data type is sufficiently high for the level of data it contains (Fig. 4).

Using the notation \((x_1, \ldots, x_n)\sigma = (x_1\sigma, \ldots, x_n\sigma)\), substitution typing can be reduced to typing of lists of terms: Suppose \( \vdash \Gamma \) and \( \vdash \Delta \). We write \( \Gamma \vdash \sigma : \Delta \) for \( \text{dom}(\sigma) = \Delta \) and \( \Gamma \vdash \hat{\Delta} \sigma : \hat{\Delta} \). Likewise, we write \( \Gamma \vdash \sigma = \sigma' : \Delta \) for \( \Gamma \vdash \hat{\Delta} \sigma = \hat{\Delta} \sigma' : \Delta \).

Definitional equality \( \Gamma \vdash u = u' : A \) is induced by rewriting function applications according to the function clauses. It is the least typed congruence over the axiom:

\[
\text{clause} \quad \Delta \vdash f \bar{q} \iff v : B \in \Sigma \quad \Gamma \vdash \sigma : \Delta
\]

\[
\Gamma \vdash f \bar{q}\sigma = v\sigma : B\sigma
\]

If \( f \bar{q} \iff v \) is a defining clause of function \( f \), then each instance arising from a well-typed substitution \( \sigma \) is a valid equation. The full list of congruence and equivalence rules is given in the long version of this paper, together with congruence rules for applications and lists of terms. As
\[ \Sigma \vdash Z \] Snipped \( Z \) is well-formed in signature \( \Sigma \).

\[ \Sigma \vdash \Delta \quad \text{data } D \Delta : \text{Set}_\ell \quad \Sigma ; \Delta \vdash \Delta_c \]

\[ \Sigma \vdash \text{constructor } c \Delta_c : D \Delta \]

\[ \Sigma \vdash \Delta \quad \text{record } R \Delta : \text{Set}_\ell \quad \Sigma ; \Delta \vdash A : \text{Set}_{\ell'} \quad \ell' \leq \ell \]

\[ \Sigma \vdash \text{projection } x : R \Delta \vdash \pi : A \]

\[ \Sigma \vdash \Delta \quad \text{definition } f : A \Delta \vdash [\bar{q}] : B \quad \Delta \vdash v : B \]

\[ \Sigma \vdash \text{clause } \Delta \vdash f \bar{q} \leftrightarrow v : B \]

\[ \Sigma_0 \subseteq \Sigma \] Signature \( \Sigma \) is a valid extension of \( \Sigma_0 \).

\[ \Sigma_0 \subseteq \Sigma \quad \Sigma \vdash Z \quad \Sigma, Z \text{ defined} \]

\[ \Sigma_0 \subseteq \Sigma \quad \Sigma \vdash Z \quad \Sigma_0 \subseteq \Sigma, Z \]

Fig. 4. Rules for well-formed signature snippets and extension.

usual in dependent type theory, definitional equality on types \( \Gamma \vdash A = B : \text{Set}_\ell \) is used for type conversion.

### 3.3 Signature well-formedness

A signature \( \Sigma \) extends \( \Sigma_0 \) if we can go from \( \Sigma_0 \) to \( \Sigma \) by adding valid snippets \( Z \), i.e. new datatypes, record types, and defined constants, but new constructors/projections/clauses only for not yet completed definitions in \( \Sigma \). A signature \( \Sigma \) is well-formed if it is a valid extension of the empty signature \( \epsilon \). Formally, we define signature extension \( \Sigma_0 \subseteq \Sigma \) via snippet typing \( \Sigma \vdash Z \) by the rules in Fig. 4, and signature well-formedness \( \vdash \Sigma \) as \( \epsilon \subseteq \Sigma \). Recall that the rules for extending the signature with a constructor (resp. projection or clause) can only be used when the corresponding data type (resp. record type or definition) is the last thing in the signature, by definition of extending the signature with a snippet \( \Sigma, Z \). When adding a constructor or projection, it is ensured that the stored data is not too big in terms of universe level \( \ell \); this preserves predicativity. However, the parameters \( \Delta \) of a data or record type of level \( \ell \) can be big, they may exceed \( \ell \).

All typing and equality judgements are monotone in the signature, thus, remain valid under signature extensions.

**Lemma 7 (Signature extension preserves inferences).** If \( \Sigma ; \Gamma \vdash u : A \) and \( \Sigma \subseteq \Sigma' \) then also \( \Sigma'; \Gamma \vdash u : A \) (and likewise for other judgements).

**Remark 8 (Coverage).** The rules for extending a signature with a function definition given by a list of clauses are not strong enough to guarantee the usual properties of a language such as type preservation and progress. For example, we could define a function with no clauses at all (violating progress), or we could add a clause where all patterns are forced patterns (violating type preservation). We prove type preservation and progress only for functions that correspond to a well-typed case tree as defined in Sect. 4.
\(\Sigma \vdash u \downarrow w\) (\(\Sigma\) fixed, dropped from rules.)

\[
\begin{align*}
\text{clause} & \quad \Delta \vdash f \bar{q} \leftrightarrow v : A \in \Sigma \quad \left[\bar{e} / \bar{q}\right] \downarrow \sigma \\
\text{w} & \downarrow \text{w} \\
\text{f e} & \downarrow \text{w}
\end{align*}
\]

Fig. 5. Rules for weak-head normalization.

\(\Sigma \vdash [v / p] \downarrow \sigma_{\perp}\) (\(\Sigma\) fixed, dropped from rules.)

\[
\begin{align*}
&\quad [v / x] \downarrow [v / x] \\
&\quad [v / [u]] \downarrow \{} \\
&\quad [v / [\text{refl}]] \downarrow \{} \\
&\quad [v / \bar{c} \bar{u}] \downarrow \sigma_{\perp} \\
&\quad [v / [c_1] \bar{p}] \downarrow \sigma_{\perp} \\
&\quad [v / [c_2] \bar{u}] \downarrow \sigma_{\perp} \\
&\quad [v / [\bar{c} \bar{p}]] \downarrow \perp \\
&\quad [\bar{p} / [\bar{c} \bar{p}]] \downarrow \perp \\
&\quad [\bar{e} / [\bar{c} \bar{p}]] \downarrow \perp
\end{align*}
\]

Fig. 6. Rules for the pattern matching and mismatching algorithm.

### 3.4 Pattern matching and evaluation rules

Evaluation to weak-head normal form \(\Sigma \vdash u \downarrow w\) is defined inductively in Fig. 5. Since our language does not contain syntax for lambda abstraction, there is no rule for \(\beta\)-reduction. Almost all terms are their own weak-head normal form; the only exception are applications \(f \bar{e}\).

Evaluation is mutually defined with matching against (co)patterns \(\Sigma \vdash [\bar{e} / \bar{q}] \downarrow \sigma_{\perp}\) (Fig. 6).

Herein, \(\sigma_{\perp}\) is either a substitution \(\sigma\) with \(\text{dom}(\sigma) = \text{PV}(\bar{q})\) or the error value \(\perp\) for mismatch. Join of lifted substitutions \(\sigma_{\perp} \uplus \tau_{\perp}\) is \(\perp\) if one of the operands is \(\perp\), otherwise the join \(\sigma \uplus \tau\).

A pattern variable \(x\) matches any term \(v\), producing singleton substitution \([v / x]\). Likewise for a forced pattern \([u]\), but it does not bind any pattern variables. Projections \(\pi\) only match themselves, and so do constructors \(c \bar{p}\), but they require evaluation \(\sigma \downarrow c \bar{u}\) of the scrutinee \(v\) and subsequent successful matching \([\bar{u} / \bar{p}] \downarrow \sigma\) of the arguments. For forced constructors \([c_1] \bar{p}\), the constructor equality test is skipped, as it is ensured by typing. Constructor \((c_1 \neq c_2)\) and projection \((\pi_1 \neq \pi_2)\) mismatches produce \(\perp\). We do not need to match against the absurd pattern; user clauses with absurd matches are never added to the signature. Recall that absurd patterns are not contained in clauses of the signature, thus, we need not consider them in the matching algorithm. Evaluating a function that eliminates absurdity will be stuck for lack of matching clauses.

A priori, matching can get stuck, if none of the rules apply. In particular, this happens when we try to evaluate an underapplied function or an open term, i.e. a term with free variables. For the purpose of the evaluation judgement, we would not need to track definite mismatch (\(\perp\)) separately from getting stuck. However, for the first-match semantics [Augustsson 1985] we do: There, a
function should reduce with the first clause that matches while all previous clauses produce a mismatch. If matching a clause is stuck, we must not try the next one.

The first-match semantics is also the reason why either $\Sigma \vdash [e / q] \perp$ or $\Sigma \vdash [\tilde{e} / \tilde{q}] \perp$ alone is not sufficient to derive $\Sigma \vdash [e \tilde{e} / q \tilde{q}] \perp$, i.e. mismatch does not dominate stuckness, nor does it short-cut matching. Suppose a function $\Sigma$ defined by the clauses $\text{true} \leftarrow \text{true}$ and $x \ y \leftarrow \text{false}$. If mismatch dominated stuckness, then both open terms $\text{false}$ and $x \text{false}$ would reduce to $\text{false}$. However, there is no case tree that accomplishes this. We have to split on the first or the second variable; either way, one of the two open terms will be stuck. We cannot even decree left-to-right splitting: see Example 5 for a definition that is impossible to elaborate to a case tree using a left-to-right splitting order. Thus, we require our pattern match semantics to be faithful with any possible elaboration of clauses into case trees.\footnote{In a sense, this is opposite to \textit{lazy pattern matching} [Maranget 1992], which aims to find the right clause with the least amount of matching.}

### 3.5 Other language features

In comparison to dependently typed programming languages like Agda and Idris, our core language seems rather reduced. In the following, we discuss how some popular features could be translated to our core language.

**Lambda abstractions and $\eta$-equality:** A lambda abstraction $\lambda x. \ t$ in context $\Gamma$ can be lifted to the top-level and encoded as auxiliary function $f : \Gamma \ x \leftarrow t$. We obtain extensionality ($\eta$) by adding the following rule to definitional equality:

$$\frac{\Gamma \vdash t_1 : (x : A) \rightarrow B \quad \Gamma \vdash t_2 : (x : A) \rightarrow B \quad \Gamma(x : A) \vdash t_1 x = t_2 x : B}{\Gamma \vdash t_1 = t_2 : (x : A) \rightarrow B \quad x \notin \text{dom}(\Gamma)}$$

**Record expressions:** Likewise, a record value $\text{record}(\pi = \bar{\sigma})$ in $\Gamma$ can be turned into an auxiliary definition by copattern matching with clauses $(f : \Gamma.\pi_i \leftarrow \nu_i)$. We could add an $\eta$-law that considers two values of record type $R$ definitionally equal if they are so under each projection of $R$. However, to maintain decidability of definitional equality, this should only applied to non-recursive records, as recursive records model coinductive types which do not admit $\eta$.

**Indexed datatypes** can be defined as regular (parameterized) datatypes with extra arguments to each constructor containing equality proofs for the indices. For example, $\text{Vec} \ A \ n$ can be defined as follows:

```haskell
data Vec (A : Set)(n : ℕ) : Set where
  nil   : n ≡_ℕ zero → Vec A n
  cons  : (m : ℕ)(x : A)(xs : Vec A m) → n ≡_ℕ suc m → Vec A n
```

**Indexed record types** can be defined analogously to indexed datatypes. For example, $\text{Vec} \ A \ n$ can also be defined as a record type:

```haskell
record Vec (A : Set)(n : ℕ) : Set where
  head : (m : ℕ) → n ≡_ℕ suc m → A
  tail : (m : ℕ) → n ≡_ℕ suc m → Vec A m
```

The ‘constructors’ nil and cons are then defined by

\begin{align*}
\text{nil} & : \text{Vec} \ A \ \text{zero} \\
\text{nil}.\text{head} \ m \ \text{∅} & = \text{impossible} \\
\text{nil}.\text{tail} \ m \ \text{∅} & = \text{impossible} \\
\text{cons} & : (n : \mathbb{N})(x : A)(xs : \text{Vec} \ A \ n) → \text{Vec} \ A \ (\text{suc} \ n) \\
\text{cons} \ n \ x \xs .\text{head} \ [n] \ \text{refl} & = x \\
\text{cons} \ n \ x \xs .\text{tail} \ [n] \ \text{refl} & = xs
\end{align*}

\footnote{In a sense, this is opposite to \textit{lazy pattern matching} [Maranget 1992], which aims to find the right clause with the least amount of matching.}
Mutual recursion can be simulated by nested recursion as long as we do not define checks for positivity and termination.

Wildcard patterns can be written as variable patterns with a fresh name. Note that an unused variable may stand for either a wildcard or a forced pattern. In the latter case our algorithm treats it as a let-bound variable in the right-hand side of the clause.

Record patterns would make sense for inductive records with $\eta$. Without changes to the core language, we can represent them by first turning deep matching into shallow matching, along the lines of Setzer et al. [2014], and then turn record matches on the left-hand side into projection applications on the right-hand side.

This concludes the presentation of our core language.

4 CASE TREES

From a user perspective it is nice to be able to define a function by a list of clauses, but for a core language this representation of functions leaves much to be desired: it is hard to see whether a set of clauses is covering all cases [Coquand 1992], and evaluating the clauses directly can be slow for deeply nested patterns [Cardelli 1984]. Recall that for type-checking dependent types, we need to decide equality of open terms which requires computing weak head normal forms efficiently.

Thus, instead of using clauses, we represent functions by a case tree in our core language. In this section, we give a concrete syntax for case trees and give typing and evaluation rules for them. We also prove that a function defined by a case tree enjoys good properties such as type preservation and coverage.

\[
Q ::= \begin{align*}
& u & \text{branch body (splitting done)} \\
| & \lambda x. \, Q & \text{bringing next argument into scope as } x \\
| & \text{record}[\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n] & \text{splitting result by possible projections } (n \geq 0) \\
| & \text{case}_x[c_1 \, \tilde{A}_1 \mapsto Q_1; \ldots; c_n \, \tilde{A}_n \mapsto Q_n] & \text{splitting on data } x (n \geq 0) \\
| & \text{case}_x[\text{ref} \mapsto r \, Q] & \text{matching on equality proof } x
\end{align*}
\]

(28)

Note that empty case and empty record are allowed, to cover the empty data type and the unit type, i.e. the record without fields.

Remark 9 (Focusing). Case trees allow us to introduce functions and records, and eliminate data. In the sense of focusing, this corresponds to the invertible rules for implication, additive conjunction, and multiplicative disjunction. (See typing rules in Fig. 7.)

4.1 Case tree typing

A case tree $Q$ for a defined constant $f : A$ is well-typed in environment $\Sigma$ if $\Sigma \vdash f := Q : A \rightarrow \Sigma'$. In this judgement, $\Sigma$ is the signature in which case tree $Q$ for function $f : A$ is well-typed, and $\Sigma'$ is the output signature which is $\Sigma$ extended with the function clauses corresponding to case tree $Q$. Note that the absence of a local context $\Gamma$ in this proposition implies that we only use case trees for top-level definitions.\(^\text{10}\)

Case tree typing is established by the generalized judgement $\Sigma; \Gamma \vdash f \, \bar{q} := Q : A \rightarrow \Sigma'$ (Fig. 7) that considers a case tree $Q$ for the instance $f \, \bar{q}$ of the function in a context $\Gamma$ of the pattern variables of $\bar{q}$. We have the following rules for $\Sigma; \Gamma \vdash f \, \bar{q} := Q : A \rightarrow \Sigma'$:

\(^{10}\)It would also be possible to embed case trees into our language as terms instead, as is the case in many other languages. We refrain from doing so in this paper for the sake of simplicity.
$\Sigma; \Gamma \vdash f \ q := Q : C \leadsto \Sigma'$

Presupposes: $\Sigma; \Gamma \vdash f [\ q ] : C$ and dom($\Gamma$) = PV($\ q$).

Checks case tree $Q$ and outputs an extension $\Sigma'$ of $\Sigma$ by the clauses represented by "$f \ q \leftarrow Q$".

$$\frac{\Sigma; \Gamma \vdash v : C}{\Sigma; \Gamma \vdash f \ q := v : C \leadsto \Sigma_i \ (\text{clause } \Gamma \vdash f \ q \leftarrow v : C)}$$

\text{CtDone}

$$\Sigma; \Gamma \vdash C = (x : A) \rightarrow B : \text{Set}_\ell$$

$$\frac{\Sigma; \Gamma \vdash f \ q x := Q : B \leadsto \Sigma'}{\Sigma; \Gamma \vdash f \ q := \lambda x. Q : C \leadsto \Sigma'}$$

\text{CtIntro}

$$\frac{\Sigma; \Gamma \vdash C = R \ \bar{o} : \text{Set}_\ell \quad \text{record self : R } \Delta : \text{Set}_\ell \quad \text{where } \pi_i : A_i \in \Sigma_0\quad \sigma = [\bar{o} / \Delta, f [\ q ] / \text{self}]}{\Sigma; \Gamma \vdash f \ q := \text{record}([\pi_1 \mapsto Q_1; \ldots ; \pi_n \mapsto Q_n] : C \leadsto \Sigma_n)}$$

\text{CtCsplit}

$$\frac{\Sigma_0; \Gamma_1 \vdash A = D \ \bar{o} : \text{Set}_\ell\quad \text{data} D \ \Delta : \text{Set}_\ell \quad \text{where } c_i A_i \in \Sigma_0\quad (\Delta'_i = \Delta_i [\bar{o} / \Delta])_{i=1 \ldots n}\quad (\rho_i = \Gamma_1 \uplus [c_i \ A'_i / x])_{i=1 \ldots n}\quad (\rho'_i = \rho_i \uplus \Gamma_1)_{i=1 \ldots n}\quad (\Sigma_{i-1}; \Gamma_1 \Delta_1 (\Gamma_1 \rho_i) + f \ q \rho'_i := Q_i : C \rho'_i \leadsto \Sigma_i)_{i=1 \ldots n}}{\Sigma_0; \Gamma_1 (x : A) \Gamma_2 \vdash f \ q := \text{case}_x(c_1 \ A'_1 \mapsto Q_1; \ldots ; c_n \ A'_n \mapsto Q_n) : C \leadsto \Sigma_n}$$

\text{CtCsplitCon}

$$\frac{\Sigma; \Gamma_1 \vdash A = (u \equiv_B v) : \text{Set}_\ell\quad \Sigma; \Gamma_1 \vdash x u =? v : B \Rightarrow \text{YES}(\Gamma'_1, \rho, \tau)}{\rho' = \rho \uplus \Gamma_1 \quad \tau' = \tau \uplus \Gamma_1\quad \Sigma; \Gamma'_1 (\Gamma_1 \rho) + f \ q \rho' := Q : C \rho' \leadsto \Sigma'}$$

\text{CtCsplitEq}

$$\frac{\Sigma; \Gamma_1 \vdash A = (u \equiv_B v) : \text{Set}_\ell\quad \Sigma; \Gamma_1 \vdash x u =? v : B \Rightarrow \text{NO}}{\Sigma; \Gamma_1 (x : A) \Gamma_2 \vdash f \ q := \text{case}_x() : C \leadsto \Sigma}$$

\text{CtCsplitAbsurdEq}

Fig. 7. The typing rules for case trees.

\textbf{CtDone} A leaf of a case tree consists of a right-hand side $v$ which needs to be of the same type $C$ of the corresponding left-hand side $f \ q$ and may only refer to the pattern variables $\Gamma$ of $\ q$.

If this is the case, the clause $f \ q \leftarrow v$ is added to the signature.

\textbf{CtIntro} If the left-hand side $f \ q$ is of function type $(x : A) \rightarrow B$ we can extend it by variable pattern $x$. The corresponding case tree is function introduction $\lambda x. Q$.

\textbf{CtCsplit} If the left-hand side is of record type $R \ \bar{o}$ with projections $\pi_i$, we can do result splitting and extend it by copattern $\pi_i$ for all $i$. We have record$([\pi_1 \mapsto Q_1; \ldots ; \pi_n \mapsto Q_n]$ (where $n \geq 0$) as the corresponding case tree, and we check each sub tree $Q_i$ for left-hand side $f \ q \ \pi_i$ in the signature $\Sigma_{i-1}$ which includes the clauses for the branches $j < i$. Note that these previous clauses may need to be checked for the current case, since we have dependent records (\textit{Example 2}).

\textbf{CtCsplitCon} If left-hand side $f \ q$ contains a variable $x$ of data type $D \ \bar{o}$, we can split on $x$ and consider all alternatives $c_i$; the corresponding case tree is case$_x(c_1 A'_1 \mapsto Q_1; \ldots ; c_n A'_n \mapsto Q_n)$. The branch $Q_i$ is checked for a refined left-hand side where $x$ has been substituted by the new pattern variables $A'_i$. Note also the threading of signatures as in rule \textbf{CtCsplit}.

The rules \textbf{CtCsplitEq} and \textbf{CtCsplitAbsurdEq} are explained in the next section.
4.2 Unification: splitting on the identity type

To split on an equality proof \( x : u \equiv_B v \) we try to unify \( u \) and \( v \). We may either find a most general unifier (m.g.u.); then we can build a case tree \( \text{case}_x\{\rho \mapsto \} \) (rule \( \text{CrSplit} \)). We may find a disunifier and build the case tree \( \text{case}_x\{\} \) (rule \( \text{CrSplitAbsurd} \)). Finally, there might be neither a m.g.u. nor a disunifier, e.g. for equality \( y + z \equiv_B y' + z' \); then type-checking fails.

In fact, in our setting we need a refinement of m.g.u.s we call **strong unifiers**. We recall the definitions of a strong unifier and a disunifier from Cockx et al. [2016], here translated to the language of this paper and specialized to the case of a single equation:

**Definition 10** (Strong unifier). Let \( \Gamma \) be a well-formed context and \( u \) and \( v \) be terms such that \( \Gamma \vdash u, v : A \). A strong unifier \((\Gamma', \sigma, \tau)\) of \( u \) and \( v \) consists of a context \( \Gamma' \) and substitutions \( \Gamma' \vdash \sigma : \Gamma(x : u \equiv_A v) \) and \( \Gamma' \vdash \tau : \Gamma' \) such that:

\[
\begin{align*}
(1) & \text{ } \Gamma' \vdash x\sigma = \text{refl} : u\sigma \equiv_{A\sigma} v\sigma & \text{(this implies the definitional equality } \Gamma' \vdash u\sigma = v\sigma : A\sigma) \\
(2) & \text{ } \Gamma' \vdash \tau ; \sigma = \text{refl} : \Gamma' \\
(3) & \text{ } \Gamma' \vdash \sigma_0 : \Gamma(x : u \equiv_A v) \text{ and } \Gamma_0 \vdash x\sigma_0 = \text{refl} : u\sigma_0 \equiv_{A\sigma_0} v\sigma_0, \text{ we have } \Gamma_0 \vdash \sigma ; \tau ; \sigma_0 = \sigma_0 : \Gamma(x : u \equiv_A v).
\end{align*}
\]

**Definition 11** (Disunifier). Let \( \Gamma \) be a well-formed context and \( \Gamma \vdash u, v : A \). A disunifier of \( u \) and \( v \) is a function \( \Gamma \vdash f : (u \equiv_A v) \rightarrow \bot \) where \( \bot \) is the empty type.

Since we use the substitution \( \sigma \) for the construction of the left-hand side of clauses, we require unification to output not just a substitution but a **pattern substitution** \( \rho \). The only properly matching pattern in \( \rho \) is \( x\rho = \text{refl} \); all the other patterns \( y\rho \) are either a forced pattern \( \{t\} \) (if unification instantiates \( y \) with \( t \)) or the variable \( y \) itself (if unification leaves \( y \) untouched).

We thus assume we have access to a proof relevant unification algorithm specified by the following judgements:

\[
\begin{align*}
\Sigma ; \Gamma \vdash x = ? y : A \Rightarrow \text{yes}(\Gamma', \rho, \tau) & \text{ ensures that } x\rho = \text{refl} \text{ and the triple } (\Gamma', [\rho], \tau) \text{ is a strong unifier. Additionally, } \Gamma' \subseteq \Gamma, \ y\tau = y \text{ and } y\rho = y \text{ for all } y \in \Gamma', \text{ and } y\rho \text{ is a forced pattern for all variables } y \in \Gamma' \\
\Sigma ; \Gamma \vdash u = ? v : A \Rightarrow \text{no} & \text{ ensures that there exists a disunifier of } u \text{ and } v.
\end{align*}
\]

**Remark 12.** During the unification of \( u \) with \( v \), each step either instantiates one variable from \( \Gamma \) (e.g. the solution step) or leaves it untouched (e.g. the injectivity step). We thus have the invariant that the variables in \( \Gamma' \) form a subset of the variables in \( \Gamma \). In effect, the substitution \( \tau \) makes the variables instantiated by unification go ‘out of scope’ after a match on \( \text{refl} \). This property ceases to hold in a language with \( \eta \)-equality for record types and unification rules for \( \eta \)-expanding a variable such as the ones given by Cockx et al. [2016]. In particular, \( \tau \) may contain not only variables but also projections applied to those variables.

4.3 Operational semantics

If a function \( f \) is defined by a case tree \( Q \), then we can compute the application of \( f \) to eliminations \( \hat{e} \) via the judgement \( \Sigma \vdash Q\sigma \hat{e} \rightarrow v \) (Fig. 8) with \( \sigma = [] \). The substitution \( \sigma \) acts as an accumulator, collecting the values for each of the variables introduced by a \( \lambda \) or by the constructor arguments in a \( \text{case}_x\{\ldots\} \). In particular, when evaluating a case tree of the form \( \text{case}_x\{\text{refl} \mapsto^{r} Q\} \), the substitution \( \tau \) is used to remove any bindings in \( \sigma \) that correspond to forced patterns.
\[ \Sigma \vdash Q \sigma \bar{e} \longrightarrow v \]

\[ \Sigma \vdash \varphi \sigma \bar{e} \longrightarrow \varphi \sigma \bar{e} \]

\[ \Sigma \vdash Q(\sigma \cup [u / x]) \bar{e} \longrightarrow v \]

\[ \Sigma \vdash (\lambda x. Q)\sigma \ u \bar{e} \longrightarrow v \]

\[ \Sigma \vdash (\lambda x. \sigma \cup [u / x]) \bar{e} \longrightarrow v \]

\[ \Sigma \vdash (\sigma \cup \text{record}(\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n)) \sigma \bar{e} \longrightarrow v \]

\[ \Sigma \vdash x\sigma \ \backslash \ c_i \ u \rightarrow \Sigma \vdash Q_i(\sigma \cup [u / \hat{A}_i]) \bar{e} \longrightarrow v \]

\[ \Sigma \vdash (\text{case}_x(c_1 \hat{A}_1 \mapsto Q_1; \ldots; c_n \hat{A}_n \mapsto Q_n))\sigma \bar{e} \longrightarrow v \]

\[ \Sigma \vdash x\sigma \ \backslash \ \text{refl} \rightarrow \Sigma \vdash (\sigma \cup \tau) \bar{e} \longrightarrow v \]

4.4 Properties

If a function \( f \) is defined by a well-typed case tree, then it enjoys certain good properties such as type preservation and coverage. The goal of this section is to state these properties, the corresponding proofs can be found in the long version of this paper. First, we need some basic lemmata.

**Lemma 13 (Well-typed case trees preserve signature well-formedness).** Let \( \Sigma \) be a well-formed signature with definition \( f : A \) where \( \overrightarrow{\text{cls}} \) the last declaration in \( \Sigma \) and let \( Q \) be a case tree such that \( \Sigma; \Gamma \vdash f \bar{q} := Q : C \rightsquigarrow \Sigma' \) where \( \Sigma \vdash \Gamma \) and \( \Sigma; \Gamma \vdash f : A \vdash \bar{q} : C \). Then \( \Sigma' \) is also well-formed.

**Proof.** By induction on \( \Sigma; \Gamma \vdash f \bar{q} := Q : C \rightsquigarrow \Sigma' \).

The following lemma implies that once the typechecker has completed checking a definition, we can replace the clauses of that definition by the case tree. This gives us more efficient evaluation of the function and guarantees that evaluation is deterministic.

**Lemma 14 (Simulation lemma).** Consider a case tree \( Q \) such that \( \Sigma_0; \Gamma \vdash f \bar{q} := Q : C \rightsquigarrow \Sigma \), let \( \sigma \) be a substitution with domain the pattern variables of \( \bar{q} \), and let \( \bar{e} \) be some eliminations. If \( \Sigma \vdash Q \sigma \bar{e} \longrightarrow t \) then there is some pattern substitution \( \rho \) and copatterns \( \bar{q}' \) such that clause \( \Delta \vdash f \bar{q} \bar{p} \bar{q}' \rightsquigarrow v : A \) is in \( \Sigma \setminus \Sigma_0 \) and \( t = v \theta \bar{e}_2 \) where \( \Sigma \vdash [\sigma \bar{e}_1 / \bar{q} \bar{p} \bar{q}'] \ \backslash \ \varnothing \theta \) and \( \bar{e} = \bar{e}_1 \bar{e}_2 \).

Conversely, any clause in \( \Sigma \setminus \Sigma_0 \) is of the form clause \( \Delta \vdash f \bar{q} \bar{p} \bar{q}' \rightsquigarrow v : A \), and for any \( \sigma \) and \( \bar{e}_1 \) and \( \bar{e}_2 \) such that \( \Sigma \vdash [\sigma \bar{e}_1 / \bar{q} \bar{p} \bar{q}'] \ \backslash \ \varnothing \theta \) we have \( \Sigma \vdash Q \sigma \bar{e}_1 \bar{e}_2 \longrightarrow v \theta \bar{e}_2 \).

**Proof.** By induction on \( Q \).

Before adding a clause \( f \bar{q} \rightsquigarrow v \) to the signature, we have to make sure that the copatterns \( \bar{q} \) only use forced patterns in places where it is justified: otherwise we might have \( \Sigma \vdash [\bar{e} / \bar{q}] \ \backslash \ \varnothing \sigma \) but \( [\bar{q}] \sigma \neq \bar{e} \). This is captured in the notion of a respectful pattern [Goguen et al. 2006]. Here we generalize their definition to the case where we do not yet know that all reductions in the signature are necessarily type-preserving.

**Definition 15.** A signature \( \Sigma \) is respectful for \( \Sigma \vdash u \ \backslash \ w \) if \( \Sigma; \Gamma \vdash u : A \) implies \( \Sigma; \Gamma \vdash u = w : A \). A signature \( \Sigma \) is respectful if it is respectful for all derivations of \( \Sigma \vdash u \ \backslash \ w \).

In particular, this means \( \Sigma; \Gamma \vdash w : A \), so evaluation with signature \( \Sigma \) is type preserving. It is immediately clear that the empty signature is respectful, since it does not contain any clauses.

**Definition 16 (Respectful copatterns).** Let \( \bar{q} \) be a list of copatterns such that \( \Sigma; \Delta \vdash [\bar{q}] : C \) where \( u \) and \( A \) are closed (i.e. do not depend on \( \Delta \)). We call \( \bar{q} \) respectful in signature \( \Sigma \) if the following holds: for any signature extension \( \Sigma \subseteq \Sigma' \) and any eliminations \( \Sigma'; \Gamma \vdash u : A \bar{e} : C \) such that

Σ' ⊢ [e / q] \( \downarrow \) σ and Σ' is respectful for any \( \Sigma' \vdash s \downarrow t \) used in the derivation of \( \Sigma' \vdash [e / q] \downarrow \sigma \), we have \( \Sigma'; \Gamma \mid u : A \vdash \bar{q}\sigma = \bar{e} : C \).

Being respectful is stable under signature extension by definition: if \( \bar{q} \) is respectful in \( \Sigma \) and \( \Sigma \subseteq \Sigma' \), then \( \bar{q} \) is also respectful in \( \Sigma' \).

**Lemma 17** (Signatures with respectful clauses are respectful). If \( \Sigma \) is a well-formed signature such that all clauses in \( \Sigma \) have respectful copatterns in \( \Sigma \), then \( \Sigma \) is respectful.

**Proof.** By induction on the derivation of \( \Sigma \vdash u \downarrow \nu \). \( \square \)

**Lemma 18** (Well-typed case trees have respectful clauses). Consider a respectful signature \( \Sigma_0 \) and a case tree \( Q \) such that \( \Sigma_0; \Gamma \vdash f \bar{q} := Q : C \leadsto \Sigma \) and \( \bar{q} \) is respectful in \( \Sigma_0 \). Then all clauses in \( \Sigma \setminus \Sigma_0 \) have respectful patterns in \( \Sigma \).

**Proof.** By induction on the derivation of \( \Sigma_0; \Gamma \vdash f \bar{q} := Q : C \leadsto \Sigma \). \( \square \)

**Theorem 19** (Type preservation). If all functions in a signature \( \Sigma \) are given by well-typed case trees, then \( \Sigma \) is respectful.

**Proof.** This is a direct consequence of the previous two lemmata. \( \square \)

**Definition 20.** A term \( u \) is normalising in a signature \( \Sigma \) if \( \Sigma \vdash u \downarrow w \), and additionally, if \( w = c \bar{v} \) then all \( \bar{v} \) are also normalising.

An elimination \( e \) is normalising if it is either a projection \( \pi \) or a normalising term \( u \). A substitution \( \sigma \) is normalising if \( x\sigma \) is normalising for all variables \( x \in \text{dom}(\sigma) \).

The definition of a normalising term (and the proof of the following lemma) would be somewhat more complicated for a language with eta-equality for record types, such as the one used by Cockx et al. [2016]. In particular, all projections of a normalising expression of record type should also be normalising.

**Lemma 21.** Suppose \( \Sigma; \Gamma \vdash x \Rightarrow v : A \Rightarrow \text{yes}(\Gamma', \rho, \tau) \) and \( \Sigma \vdash \sigma_0 : \Gamma \) such that \( \Sigma \vdash u\sigma_0 = v\sigma_0 : A\sigma_0 \). If \( \sigma_0 \) is normalising, then so is \( \tau; \sigma_0 \).

**Proof.** By construction, \( y\tau = y \) for each variable \( y \) in \( \Gamma' \) (see Remark 12), so it follows trivially that \( \tau; \sigma_0 \) is normalising. \( \square \)

**Theorem 22** (Coverage). Let \( Q \) be a case tree such that \( \Sigma_0; \Gamma \vdash f \bar{q} := Q : C \leadsto \Sigma \). Let further \( \Sigma \vdash \sigma_0 : \Gamma \) be a (closed) substitution and \( \Sigma \mid f \bar{q}\sigma_0 : C\sigma_0 \vdash \bar{e} : B \) be (closed) eliminations such that \( \sigma_0 \) and \( \bar{e} \) are normalising in \( \Sigma \) and \( B \) is not definitionally equal to a function type or a record type. Then \( \Sigma \vdash Q\sigma_0 \bar{e} \rightarrow v \) for some \( v \).

**Proof.** By induction on the case tree \( Q \). \( \square \)

In particular, this theorem tells us that if \( \Sigma_0 \vdash f := Q : C \leadsto \Sigma \) and the eliminations \( \Sigma \mid f : C \vdash \bar{e} : A \) are normalising, then \( \Sigma \vdash Q[] \bar{e} \rightarrow v \). Thus evaluation of a function defined by a well-typed case tree applied to closed arguments can never get stuck.

### 5 ELABORATION

In the previous two sections, we have described a core language with inductive datatypes, coinductive records, identity types, and functions defined by well-typed case trees. On the other hand, we also have a surface language consisting of declarations of datatypes, record types, and functions by dependent (co)pattern matching. In this section we show how to elaborate a program in this surface language to a well-formed signature in the core language.
\[
\Sigma \vdash decl \leadsto \Sigma'
\]
Presupposes: \(\vdash \Sigma\). Entails: \(\vdash \Sigma'\).

\[
\Sigma \vdash \Delta \quad (\Sigma, \text{data } D \Delta : \text{Set}_\ell) \mid D \Delta : \text{Set}_\ell \vdash \overline{\text{con}} \leadsto \Sigma'
\]
\[
\Sigma \vdash (\text{data } D \Delta : \text{Set}_\ell \text{ where } \overline{\text{con}}) \leadsto \Sigma'
\]

\[
\Sigma \vdash \Delta \quad (\Sigma, \text{record } R \Delta : \text{Set}_\ell) \mid \text{self } : R \Delta : \text{Set}_\ell \vdash \overline{\text{field}} \leadsto \Sigma'
\]
\[
\Sigma \vdash (\text{record } \text{self } : R \Delta : \text{Set}_\ell \text{ where } \overline{\text{field}}) \leadsto \Sigma'
\]

\[
\Sigma \vdash A \quad (\Sigma, \text{definition } f : A) \vdash P \mid f := Q : A \leadsto \Sigma'
\]
\[
\Sigma \vdash (\text{definition } f : A \text{ where } P) \leadsto \Sigma'
\]

\[
\Sigma \mid D \Delta : \text{Set}_\ell \vdash \overline{\text{con}} \leadsto \Sigma'
\]
Presupposes: \(\vdash \Sigma\) and \(\Sigma \vdash \Delta : \Lambda \rightarrow \text{Set}_\ell\). Entails: \(\vdash \Sigma'\).

\[
\Sigma ; \Delta \vdash \ell \quad (\Sigma, \text{constructor } c \Delta_c : D \Delta) \mid D \Delta : \text{Set}_\ell \vdash \overline{\text{con}} \leadsto \Sigma'
\]
\[
\Sigma \mid D \Delta : \text{Set}_\ell \vdash c \Delta_c, \overline{\text{con}} \leadsto \Sigma'
\]

\[
\Sigma \mid \text{self } : R \Delta : \text{Set}_\ell \vdash \overline{\text{field}} \leadsto \Sigma'
\]
Presupposes: \(\vdash \Sigma\) and \(\Sigma \vdash \Delta : \Lambda \rightarrow \text{Set}_\ell\). Entails: \(\vdash \Sigma'\).

\[
\Sigma ; \Delta (\text{self } : R \hat{\Delta}) \vdash A : \text{Set}_\ell \quad \ell' \leq \ell
\]
\[
(\Sigma, \text{projection } \text{self } : R \Delta \vdash \pi : A) \mid \text{self } : R \Delta : \text{Set}_\ell \vdash \overline{\text{field}} \leadsto \Sigma'
\]
\[
\Sigma \mid \text{self } : R \Delta : \text{Set}_\ell \vdash \pi : A, \overline{\text{field}} \leadsto \Sigma'
\]

Fig. 9. Rules for checking declarations of data types, record types, and defined symbols.

The main goal of this section is to describe the elaboration of a definition given by a set of (unchecked) clauses to a well-typed case tree, and prove that this translation (if it succeeds) preserves the first-match semantics of the given clauses. Before we dive into this, we first describe the elaboration for data and record types.

### 5.1 Elaborating data and record types

Figure 9 gives the rules for checking declarations, constructors and projections. These rules are designed to correspond closely to those for signature extension in Fig. 4. Consequentially, if \(\vdash \Sigma\) and \(\Sigma \vdash \textit{decl} \leadsto \Sigma'\), then also \(\vdash \Sigma'\).

### 5.2 From clauses to a case tree

In Section 2 we showed how our elaboration algorithm works in a number of examples, here we describe it in general. The inputs to the algorithm are the following:

- A signature \(\Sigma\) containing previous declarations, as well as clauses for the branches of the case tree that have already been checked.
- A context \(\Gamma\) containing the types of the pattern variables: \(\text{dom}(\Gamma) = \text{PV}(\bar{q})\).
- The function \(f\) currently being checked.
- The copatterns \(\bar{q}\) for the current branch of the case tree.
- The refined target type \(C\) of the current branch.
- The user input \(P\), which is described below.
The outputs of the algorithm are a signature $\Sigma'$ extending $\Sigma$ with new clauses and a well-typed case tree $Q$ such that $\Sigma; \Gamma \vdash f \bar{q} := Q : C \leadsto \Sigma'$.

We represent the user input $P$ to the algorithm as an (ordered) list of partially decomposed clauses, called a left-hand side problem or lhs problem for short. Each partially decomposed clause is of the form $[E]\bar{q} \leadsto \text{rhs}$ where $E$ is an (unordered) set of constraints $[w_k f^i p_k : A_k | k = 1 \ldots l]$ between a pattern $p_k$ and a term $w_k, \bar{q}$ is a list of copatterns, and rhs is a right-hand side. In the special case $E$ is empty, we have a complete clause written as $\bar{q} \leftarrow \text{rhs}$.

Elaboration of an lhs problem to a well-typed case tree $\Sigma; \Gamma \vdash P | f \bar{q} := Q : C \leadsto \Sigma'$ is defined in Fig. 10. This judgement is designed as an algorithmic version of the typing judgement for case trees $\Sigma; \Gamma \vdash f \bar{q} := Q : C \leadsto \Sigma'$, where the extra user input $P$ guides the construction of the case tree. Each of the rules in Fig. 10 is a refined version of one of the rules in Fig. 7, so any case tree produced by this elaboration is well-typed by construction.

To check a definition of $f : A$ with clauses $\bar{q}_i \leftarrow \text{rhs}_i$ for $i = 1, \ldots, n$, the algorithm starts with $\Gamma = e, u = f,$ and $P = [\bar{q}_i \leftarrow \text{rhs}_i | i = 1 \ldots n]$. If we obtain $\Sigma; \Gamma \vdash P | f := Q : A \leadsto \Sigma'$, then the function $f$ can be implemented using the case tree $Q$.

During elaboration, the algorithm maintains the invariants that $\vdash \Sigma$ is a well-formed signature, $\Sigma; \Gamma$ is a well-formed context, and $\Sigma; \Gamma \vdash f [\bar{q}] : C$. It also maintains the invariant that for each constraint $w_k f^i p_k : A_k$ in the lhs problem, we have $\Sigma; \Gamma \vdash w_k : A_k$.

The rules for $\Sigma; \Gamma \vdash P | f \bar{q} := Q : C \leadsto \Sigma'$ make use of some auxiliary operations for manipulating lhs problems:

- After each step, the algorithm uses $\Sigma; \Gamma \vdash E \Rightarrow \text{Solved}(\sigma)$ (Fig. 11) to check if the first user clause has no more (co)patterns, and all its constraints are solved. If this is the case, it returns a substitution $\sigma$ assigning a well-typed value to each of the user-written pattern variables.
- After introducing a new variable, the algorithm uses $P (x : A)$ (Fig. 12) to remove the first application pattern from each of the user clauses and to introduce a new constraint between the variable and the pattern.
- After a copattern split on a record type, the algorithm uses $P \pi$ (Fig. 13) to partition the clauses in the lhs problem according to the projection they belong to.
- After a case split on a datatype or an equality proof, the algorithm uses $\Sigma \vdash P \sigma \Rightarrow \Sigma'$ (Fig. 14) to refine the constraints in the lhs problem. It uses judgements $\Sigma \vdash v \bar{f} \bar{p} : A \Rightarrow E_\perp$ and $\Sigma \vdash \bar{v} \bar{f} \bar{p} : \Delta \Rightarrow E_\perp$ (Fig. 15) to simplify the constraints if possible, and to filter out the clauses that definitely do not match the current branch.
- To check an absurd pattern $\emptyset$, the algorithm uses $\Sigma; \Gamma \vdash \emptyset : A$ (Fig. 16) to ensure that the type of the pattern is a caseless type [Goguen et al. 2006], i.e. a type that is empty and cannot even contain constructor-headed terms in an open context. Our language has two kinds of caseless types: datatypes $D \bar{v}$ with no constructors, and identity types $u \equiv_A v$ where $\Sigma; \Gamma \vdash_x u =? v : A \Rightarrow \text{NO}$.

The following rules constitute the elaboration algorithm $\Sigma; \Gamma \vdash P | f \bar{q} := Q : C \leadsto \Sigma'$:

- **DONE** applies when the first user clause in $P$ has no more copatterns and all its constraints are solved according to $\Sigma; \Gamma \vdash E \Rightarrow \text{Solved}(\sigma)$. If this is the case, then construction of the case tree is finished, adding the clause $\text{clause} \Gamma \vdash f \bar{q} \leftarrow v\sigma : C$ to the signature.
- **INTRO** applies when $C$ is a function type and all the user clauses have at least one application copattern. It constructs the case tree $\lambda x. Q$, using $P (x : A)$ to construct the subtree $Q$.  

\[ \Sigma; \Gamma \vdash P \mid f \bar{q} := Q : C \leadsto \Sigma' \]

In all rules \( P = \{ [E_i] \bar{q}_i \leadsto \text{rhs}_i \mid i = 1 \ldots m \} \).

Presupposes: \( \Sigma; \Gamma \vdash f \ [\bar{q}] : C \) and \( \text{dom}(\Gamma) = \text{PV}(\bar{q}) \).

Entails: \( \Sigma; \Gamma \vdash f \bar{q} := Q : C \leadsto \Sigma' \).

\[
\frac{q_1 = e}{\Sigma; \Gamma \vdash E_1 \Rightarrow \text{solved}(\sigma)} \quad \text{rhs}_1 = v \quad \Sigma; \Gamma \vdash v \sigma : C \quad \text{DONE}
\]

\[
\frac{\Sigma; \Gamma \vdash P \mid f \bar{q} := v \sigma : C \leadsto \Sigma, \text{clause} \Gamma \vdash f \bar{q} \leadsto v \sigma : C}{\Sigma; \Gamma \vdash P \mid f \bar{q} := v \sigma : C \leadsto \Sigma}
\]

\[
\frac{q_1 \vdash C \setminus (x : A) \rightarrow B}{\Sigma; \Gamma(\langle x : A \rangle) \vdash P (\langle x : A \rangle) \mid f \bar{q} \ x := Q : B \leadsto \Sigma'} \quad \text{Intro}
\]

\[
\frac{\Sigma \vdash C \setminus (x : A) \rightarrow B}{\Sigma; \Gamma \vdash P \mid f \bar{q} := \lambda x. Q : C \leadsto \Sigma'} \quad \text{Cosplit}
\]

\[
\frac{q_1 = 0 \quad m = 1 \quad \Sigma \vdash C \setminus R \bar{v} \quad \text{record} \ _\vdash R \Delta : \text{Set}_\ell \text{ where } \ell 
\Sigma; \Gamma \vdash P \mid f \bar{q} := \text{record}\{} : C \leadsto \Sigma \quad \text{CosplitEmpty}
\]

\[
\frac{(x / \ell \ c_\ell \ p : A) \in E_1 \quad \Sigma \vdash A \setminus D \bar{v} \quad \Gamma = \Gamma_1 (\langle x : A \rangle) \Gamma_2}{\text{data} \ D : \text{Set}_\ell \text{ where } c_\ell \Delta_\ell \in \Sigma_0}
\frac{\Delta_\ell' = \Delta_1 \bar{v} / \Delta \quad \rho_1 = \Pi_{\Gamma_1} \cup [c_\ell \Delta_\ell' / x] \quad \rho_1' = \rho_1 \cup \Pi_{\Gamma_2}}{\Sigma_n \vdash P \rho_1' \Rightarrow P_1 \quad \left( \Sigma_{n-1} ; \Gamma \Delta_1' (\Gamma_2 \rho_1) \vdash P_1 ; f \bar{q} \rho_1' \Rightarrow Q_1 ; C \rho_1' \leadsto \Sigma_1 \right)_{i=1 \ldots n} \text{SplitCon}}
\frac{\Sigma_0 ; \Gamma \vdash P \mid f \bar{q} := \text{case}_x \{c_1 \Delta_1' \Rightarrow Q_1 ; \ldots ; c_n \Delta_n' \Rightarrow Q_n\} : C \leadsto \Sigma_n \text{ SplitEq}}
\frac{(x / \ell \ \text{refl} : A) \in E_1 \quad \Sigma \vdash A \setminus u \equiv_B v \quad \Gamma = \Gamma_1 (\langle x : A \rangle) \Gamma_2}{\Sigma; \Gamma \vdash P \rho' \Rightarrow P' \quad \Sigma; \Gamma (\Gamma_2 \rho') \vdash P' \mid f \bar{q} \rho' := Q : C \rho' \leadsto \Sigma' \text{ SplitEmpty}}
\frac{(x / \ell \ \emptyset : A) \in E_1 \quad \Sigma \vdash \emptyset : A \quad \text{rhs}_1 = \text{impossible}}{\Sigma; \Gamma \vdash P \mid f \bar{q} := \text{case}_x \{} : C \leadsto \Sigma \text{ SplitEmpty}}
\]

Fig. 10. Rules for checking a list of clauses and elaborating them to a well-typed case tree.

\[
\frac{\Sigma; \Gamma \vdash E \Rightarrow \text{solved}(\sigma)}{(\Sigma \vdash [w_k / p_k] \setminus \sigma_k)_{k=1 \ldots n} \quad \sigma = \bigcup_k \sigma_k \quad (\Sigma; \Gamma \vdash [p_k] \sigma = w_k : A_k)_{k=1 \ldots n}}{\Sigma; \Gamma \vdash [w_k / \ell \ p_k : A_k \mid k = 1 \ldots n] \Rightarrow \text{solved}(\sigma)}
\]

Fig. 11. Rule for constructing the final substitution and checking all constraints when splitting is done.

**Cosplit** applies when \( C \) is a record type and all the user clauses have at least one projection copattern. It constructs the case tree \( \text{record} \{ \pi_1 \mapsto Q_1 ; \ldots ; \pi_n \mapsto Q_n \} \), using \( P \ . \pi_i \) to construct the branch \( Q_i \) corresponding to projection \( \pi_i \).
Elaborating dependent (co)pattern matching

1. Replace the first application pattern \( p \) in each clause by the constraint \( x \mapsto p : A \).

\[ e \ (x : A) = e \]

\[ ([E]p \bar{q} \leftarrow \text{rhs}, \ P \ (x : A) = ([E \cup \{ x \mapsto p : A \}]q \leftarrow \text{rhs}), \ P \ (x : A) \]

**Fig. 12.** Partially decomposed clauses after introducing a new variable (partial function).

2. Keep only clauses with copattern \( .\pi \), with this copattern removed.

\[ e .\pi = e \]

\[ ([E].\pi \bar{q} \leftarrow \text{rhs}, \ P .\pi) \]

\[ ([E].\pi' \bar{q} \leftarrow \text{rhs}, \ P .\pi) = P .\pi \]

if \( \pi \neq \pi' \)

**Fig. 13.** Partially decomposed clauses after a copattern split (partial function).

3. Simplify the constraints of a partially decomposed clause.

\[ E = \{ \text{w}_k \mapsto \text{p}_k : A_k \mid k = 1 \ldots n \} \]

\[ ([\text{w}_k \sigma] \mapsto \text{p}_k : A_k \sigma \Rightarrow E_i)_{k=1 \ldots n} \]

\[ P \sigma \Rightarrow P' \]

**Fig. 14.** Rules for transforming partially decomposed clauses after refining the pattern with a case split.

4. Simplify the constraints of a partially decomposed clause.

\[ \gamma \mapsto p : A \Rightarrow E \]

**Fig. 15.** Rules for simplifying the constraints of a partially decomposed clause.

5. Rules for caseless types.

**Fig. 16.** Rules for caseless types.
COSPLITEMPTY applies when $C$ is a record type with no projections and the first clause starts with an absurd pattern. It then constructs the case tree $\text{record}\{\}$.

SPLITCON applies when the first clause has a constraint of the form $x \not\in \Sigma \bar{c}_j \cdot \rho$ and the type of $x$ in $\Gamma$ is a datatype. For each constructor $c_j$ of this datatype, it constructs a pattern substitution $\rho_i$, replacing $x$ by $c_j$ applied to fresh variables. It then constructs the case tree $\text{case}\{c_1 \Rightarrow Q_1; \ldots; c_n \Rightarrow Q_n\}$, using $\Sigma \vdash P \rho_i \Rightarrow P_i$ to construct the branches $Q_i$.

SPLITEQ applies when the first clause has a constraint of the form $x \not\in \Sigma \bar{f}$ and the type of $x$ in $\Gamma$ is an identity type $u \equiv_A \nu$. It tries to unify $u$ with $\nu$, expecting a positive success. If unification succeeds with output $(\Gamma', \rho, \tau)$, it constructs the case tree $\text{case}\{\text{refl} \Rightarrow \tau' \cdot Q\}$, using $\Sigma \vdash P \rho' \Rightarrow P'$ to construct the subtree $Q$. Here $\rho'$ and $\tau'$ are lifted versions of $\rho$ and $\tau$ over the part of the context that is untouched by unification.

SPLITEMPTY applies when the first clause has a constraint of the form $x \not\in \Sigma \emptyset$, and the type of $x$ is a caseless type according to $\Sigma; \Gamma \vdash \emptyset : A$. It then produces the case tree $\text{case}\{\}$.

Lemma 27. Let $\sigma$ be a substitution and suppose $\Sigma \vdash \bar{v} \not\in \Sigma \bar{p} : A \Rightarrow E$. Then the following hold:

- $\Sigma \vdash [\bar{v} \sigma / \bar{p}] \Downarrow \sigma'$ if and only if for each $(w_k \not\in \Sigma \bar{p}_k : B_k) \in E$, we have $\Sigma \vdash [w_k \sigma / \bar{p}_k] \Downarrow \sigma_k$, and $\sigma' = \bigcup_k \sigma_k$.
- $\Sigma \vdash [\bar{v} \sigma / \bar{p}] \Downarrow \bot$ if and only if for some $(w_k \not\in \Sigma \bar{p}_k : B_k) \in E$, we have $\Sigma \vdash [w_k \sigma / \bar{p}_k] \Downarrow \bot$.

Proof. This follows directly from the rules of matching (Fig. 6) and simplification (Fig. 15). \qed
The following lemma is the main component of the proof. It generalizes the statement of Theorem 25 to the case where the left-hand side has already been refined to \( f \bar{q} \) and the user clauses have been partially decomposed. From this lemma, the main theorem follows directly by taking \( \bar{q} = \epsilon \) and \( E_i = \{ \} \) for \( i = 1 \ldots n \).

**Lemma 28.** Let \( P = \{ [E_i] \bar{q}_i \iff \text{rhs}_i \mid i = 1 \ldots n \} \) be a list of well-scoped partially decomposed clauses such that \( \Sigma_0; \Gamma_0 \vdash P \mid f \bar{q} := Q : C \leadsto \Sigma \), and suppose \( \Gamma \vdash \sigma_0 : \Gamma_0 \) and \( \Sigma \vdash f \bar{q} \sigma_0 : C \sigma_0 \vdash \epsilon : B. \) If there is an index \( i \) such that:

- For each \( j = 1 \ldots i - 1 \) and each constraint \( (w_k / \ell_p k : A_k) \in E_j \), either \( \Sigma \vdash [w_k \sigma_0 / p_k] \downarrow \theta_{jk} \) or \( \Sigma \vdash [w_k \sigma_0 / p_k] \downarrow \bot \).
- For each \( j = 1 \ldots i - 1 \), either \( \Sigma \vdash [\bar{\epsilon} / \bar{q}_j] \downarrow \theta_{jo} \) or \( \Sigma \vdash [\bar{\epsilon} / \bar{q}_j] \downarrow \bot \).
- For each \( j = 1 \ldots i - 1 \), either \( \Sigma \vdash [w_k \sigma_0 / p_k] \downarrow \bot \) for some constraint \( (w_k / \ell_p k : A_k) \in E_j \), or \( \Sigma \vdash [w_k \sigma_0 / p_k] \downarrow \theta_k \).
- \( \Sigma \vdash [\bar{\epsilon} / \bar{q}_j] \downarrow \theta_0 \).

Then \( \text{rhs}_i = v_l \) is not impossible and \( \Sigma \vdash Q \sigma_0 \bar{\epsilon} \iff v_l \theta \) where \( \theta = \theta_0 \cup (\emptyset k) \theta_k \).

**Proof.** By induction on the derivation of \( \Sigma_0; \Gamma_0 \vdash P \mid f \bar{q} := Q : C \leadsto \Sigma \). \( \square \)

## 6 RELATED WORK

Dependent pattern matching was introduced in the seminal work by Coquand [1992]. It is used in the implementation of various dependently typed languages such as Agda [Norell 2007], Idris [Brady 2013], the Equations package for Coq [Sozeau 2010], and Lean [de Moura et al. 2015].

Previous work by Norell [2007], Sozeau [2010], and Cockx [2017] also describe elaborations from clauses to a case tree, but in much less detail than presented here, and they do not support copatterns or provide a correctness proof. In cases where both our current algorithm and these previous algorithms succeed, we expect there is no difference between the resulting case trees. However, our current algorithm is much more flexible in the placement of dot patterns, so it accepts more definitions than was possible before (see Example 4).

The translation from a case tree to primitive datatype eliminators was pioneered by McBride [2000] and further detailed by Goguen et al. [2006] for type theory with uniqueness of identity proofs and Cockx [2017] in a theory without.

Forced patterns, as well as forced constructors, were introduced by Brady et al. [2003]. Brady et al. focus mostly on the compilation process and the possibility to erase arguments and constructor tags, while we focus more on the process of typechecking a definition by pattern matching and the construction of a case tree.

Copatterns were introduced in the simply-typed setting by Abel et al. [2013] and subsequently used for unifying corecursion and recursion in System F\( \omega \) [Abel and Pientka 2013]. In the context of Isabelle/HOL, Blanchette et al. [2017] use copatterns as syntax for mixed recursive-corecursive definitions. Setzer et al. [2014] give an algorithm for elaborating a definition by mixed pattern/copattern matching to a nested case expression, yet only for a simply typed language. Thibodeau et al. [2016] present a language with deep (co)pattern matching and a restricted form of dependent types. In their language, types can only depend on a user-defined domain with decidable equality and the types of record fields cannot depend on each other, thus, a self value is not needed for checking projections. They feature indexed data and record types in the surface language which are elaborated into non-indexed types via equality types, just as in our core language.

The connection between focusing [Andreoli 1992] and pattern matching has been systematically explored by Zeilberger [2009]. In Licata et al. [2008] copatterns (“destructor patterns”) also appear.
in the context of simple typing with connectives from linear logic. Krishnaswami [2009] boils the connection to focusing down to usual non-linear types; however, he has no copatterns as he only considers the product type as multiplicative (tensor), not additive. Thibodeau et al. [2016] extend the connection to copatterns for indexed record types.

Elaborating a definition by pattern matching to a case tree [Augustsson 1985] simultaneously typechecks the clauses and checks their coverage, so our algorithm has a lot in common with coverage checking algorithms. For example, Norell [2007] views the construction of a case tree as a part of coverage checking. Oury [2007] presents a similar algorithm for coverage checking and detecting useless cases in definitions by dependent pattern matching.

7 FUTURE WORK AND CONCLUSION

In this paper, we give a description of an elaboration algorithm for definitions by dependent co-pattern matching that is at the same time elegant enough to be intuitively understandable, simple enough to study formally, and detailed enough to serve as the basis for a practical implementation.

The implementation of our algorithm as part of the Agda typechecker is at the moment of writing still work in progress. In fact, the main reason to write this paper was to get a clear idea of what exactly should be implemented. For instance, while working on the proof of Theorem 25, we were quite surprised to discover that it did not hold at first: matching was performed lazily from left to right, but the case tree produced by elaboration may not agree on this order! This problem was not just theoretical, but also manifested itself in the implementation of Agda as a violation of subject reduction [Agda issue 2018a]. Removing the shortcut rule from the definition of matching removed this behavioral divergence mismatch. The complete formalization of the elaboration algorithm in this paper lets us continue the implementation with confidence.

Agda also has a number of features that are not described in this paper, such as nonrecursive record types with η equality and general indexed datatypes (not just the identity type). The implementation also has to deal with the insertion of implicit arguments, the presence of metavariables in the syntax, and reporting understandable errors when the algorithm fails. Based on our practical experience, we are confident that the algorithm presented here can be extended to deal with all of these features.

REFERENCES


