Elaborating dependent (co)pattern matching: No pattern left behind

JFP **30**, e2, 43 pages, 2020.
doi:10.1017/S0956796819000182

Elaborating dependent (co)pattern matching: No pattern left behind

JESPER COCKX

Chalmers and Gothenburg University, Sweden (until October 2019) and TU Delft, Netherlands (from December 2019)
(*e-mail:* jesper@sikanda.be)

ANDREAS ABEL

Chalmers and Gothenburg University, Sweden
(*e-mail:* andreas.abel@gu.se)

Abstract

In a dependently typed language, we can guarantee correctness of our programmes 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 programmes 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 data types, 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 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,

[1] This paper is best viewed in colour.

our programme 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 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 data type eliminators.[2] The second step has been studied in detail and is known to preserve the semantics of the case tree precisely (Goguen *et al*., 2006; Cockx, 2017). 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 Norell (2007) and Sozeau (2010), 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 (Abel *et al*., 2013). They can be nested and mixed with regular patterns. Elaboration of definitions by copattern matching has been studied for simply typed languages by Setzer *et al*. (2014), 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 CoNat of possibly infinite natural numbers by the two projections $\mathsf{iszero} : \mathsf{Bool}$ and $\mathsf{pred} : \mathsf{iszero} \equiv_{\mathsf{Bool}} \mathsf{false} \to \mathsf{CoNat}$. We use copatterns to define the co-natural number cozero:

$$\begin{aligned} &\mathsf{cozero} : \mathsf{CoNat} \\ &\mathsf{cozero}\ .\mathsf{iszero} = \mathsf{true} \\ &\mathsf{cozero}\ .\mathsf{pred}\quad \emptyset \end{aligned} \tag{1}$$

Here, the constant cozero is being defined with the field iszero equal to true (and no value for pred).

[2] 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.

To refute the proof of $\mathsf{cozero}\ .\mathsf{iszero} \equiv_{\mathsf{Bool}} \mathsf{false}$ with an absurd pattern $\emptyset$, the typechecker needs to know already that $\mathsf{cozero}\ .\mathsf{iszero} = \mathsf{true}$, 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 *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*,[3] there are often multiple positions where to place them. For example, in the proof of symmetry of equality:

$$\begin{aligned} &\mathsf{sym} : (x\, y : A) \to x \equiv_A y \to y \equiv_A x \\ &\mathsf{sym}\ x\ \lfloor x \rfloor\ \mathsf{refl} = \mathsf{refl} \end{aligned} \tag{2}$$

it should not matter if we instead write $\mathsf{sym}\ \lfloor x \rfloor\ x\ \mathsf{refl} = \mathsf{refl}$. In fact, we even allow the apparently non-linear definition $\mathsf{sym}\ x\ x\ \mathsf{refl} = \mathsf{refl}$.

Our elaboration algorithm addresses this by treating forced patterns as *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 zero instead of $\lfloor \mathsf{zero} \rfloor$, or x instead of $\lfloor x \rfloor$.

With our elaboration algorithm, it is easy to extend the pattern syntax with *forced constructor patterns* such as $\lfloor \mathsf{suc} \rfloor\ n$ (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:

$$\begin{aligned} &\mathsf{max} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &\mathsf{max}\ \mathsf{zero}\ \ y\ \ \ \ \ \ = y \\ &\mathsf{max}\ x\ \ \ \ \mathsf{zero}\ \ = x \\ &\mathsf{max}\ (\mathsf{suc}\, x)\ (\mathsf{suc}\, y) = \mathsf{suc}\ (\mathsf{max}\, x\, y) \end{aligned} \tag{3}$$

we do not get $\mathsf{max}\, x\, \mathsf{zero} = x$ but only $\mathsf{max}\ (\mathsf{suc}\, x')\ \mathsf{zero} = \mathsf{suc}\, x'$. 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,[4] 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.

[3] Forced patterns are also called *presupposed terms* (Brady *et al*., 2003), *inaccessible patterns* (Norell, 2007), or, in Agda, *dot patterns*.

[4] Note that, in the example, the open term $\mathsf{max}\, x\, \mathsf{zero}$ does not produce a mismatch with the first clause since it could match if variable x was replaced by zero. In the first-match semantics, evaluation of $\mathsf{max}\, x\, \mathsf{zero}$ is stuck.

Contributions

- We present a dependently typed core language with inductive data types, coinductive record types, and an identity type. The language is focused (Andreoli, 1992; Zeilberger, 2008; Krishnaswami, 2009): 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. This re-implementation has been released[5] as part of Agda 2.5.4; it uncovers and fixes 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 Idris (2013), Lean (de Moura *et al*., 2015), and the Equations package for Coq (Sozeau, 2010).

Compared to the conference version of this paper (Cockx & Abel, 2018), we add the following:

- We give a more fine-grained small-step semantics for our core language, which allows us to state and prove type preservation without assuming normalization (see Theorem 17).[6]
- We add a discussion on the treatment of catch-all clauses by the algorithm described in this paper and how it differs from the implementation used by Agda (see Section 2.6).
- We include the detailed proofs that were omitted from the conference version.

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.

We start by introducing definitions by dependent (co)pattern matching and our elaboration algorithm to a case tree by a number of examples in Section 2. We then describe our core language in Section 3: the syntax, the rules for typing and equality, and the evaluation

[5] Agda 2.5.4 released on 2018/06/02, changelog: `https://hackage.haskell.org/package/Agda-2.5.4/changelog`.

[6] In most dependently typed languages, a termination checker ensures normalization of recursive functions. Likewise, a productivity checker can ensure normalization in the presence of infinite structures defined by copatterns. However, by proving preservation independently from normalization, we allow our approach to be extended to languages without such checks (such as Haskell), while at the same time making the metatheory more modular.

rules. In Section 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 progress. Finally, in Section 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. Section 6 discusses related work, and Section 7 concludes.

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 $\Gamma \vdash P \mid u : C$, called a *lhs problem*, consisting of:

- A context Γ, i.e. a list of variables annotated with types. Initially, Γ is the empty context ε.
- The current target type C. This type may depend on variables bound in Γ. Initially, C is the type of the function being defined.
- A representation of the 'self' object u, which is a term of type C in context Γ. Initially, u is the function being defined itself. In each leaf of the case tree, u will represent the left-hand side of the clause, where certain variables might be specialized due to overlap with previous clauses.
- 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 Section 5.

2.1 A first example: Maximum of two numbers

Let us define a function $\mathsf{max} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ by pattern matching as in the introduction (3). The initial lhs problem is $\vdash P_0 \mid \mathsf{max} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ where:

$$P_0 = \begin{cases} \mathsf{zero} \quad j \quad\quad\; \hookrightarrow j \\ i \quad\quad \mathsf{zero} \quad \hookrightarrow i \\ (\mathsf{suc}\, k)\ (\mathsf{suc}\, l) \hookrightarrow \mathsf{suc}\, (\mathsf{max}\, k\, l) \end{cases} \tag{4}$$

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

$$P_1 = \begin{cases} [m \mathrel{/^?} \mathsf{zero}]\ j \quad\quad\; \hookrightarrow j \\ [m \mathrel{/^?} i] \quad\quad \mathsf{zero} \quad \hookrightarrow i \\ [m \mathrel{/^?} \mathsf{suc}\, k]\ (\mathsf{suc}\, l) \hookrightarrow \mathsf{suc}\, (\mathsf{max}\, k\, l) \end{cases} \tag{5}$$

This operation strips the first user pattern from each clause and replaces it by a constraint $m \mathrel{/^?} p$ 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 SplitCon).[7] This transforms the problem into two subproblems $\vdash P_2 \mid \mathsf{max}\ \mathsf{zero} : \mathbb{N} \to \mathbb{N}$ and $(p : \mathbb{N}) \vdash P_3 \mid \mathsf{max}\ (\mathsf{suc}\ p) : \mathbb{N} \to \mathbb{N}$ where:

$$P_2 = \begin{cases} [\mathsf{zero} \mathrel{/^?} \mathsf{zero}]\ j & \hookrightarrow j \\ [\mathsf{zero} \mathrel{/^?} i] \quad \mathsf{zero} & \hookrightarrow i \\ [\mathsf{zero} \mathrel{/^?} \mathsf{suc}\ k]\ (\mathsf{suc}\ l) & \hookrightarrow \mathsf{suc}\ (\mathsf{max}\ k\ l) \end{cases} \tag{6}$$

$$P_3 = \begin{cases} [\mathsf{suc}\ p \mathrel{/^?} \mathsf{zero}]\ j & \hookrightarrow j \\ [\mathsf{suc}\ p \mathrel{/^?} i] \quad \mathsf{zero} & \hookrightarrow i \\ [\mathsf{suc}\ p \mathrel{/^?} \mathsf{suc}\ k]\ (\mathsf{suc}\ l) & \hookrightarrow \mathsf{suc}\ (\mathsf{max}\ k\ l) \end{cases} \tag{7}$$

We simplify the constraints as follows: clauses with absurd constraints (such as $\mathsf{zero} \mathrel{/^?} \mathsf{suc}\ k$) are removed; trivial constraints (such as $\mathsf{zero} \mathrel{/^?} \mathsf{zero}$) are dropped; and constraints between equal constructors (such as $\mathsf{suc}\ p \mathrel{/^?} \mathsf{suc}\ k$) are simplified (i.e. to $p \mathrel{/^?} k$):

$$P_2 = \begin{cases} j & \hookrightarrow j \\ [\mathsf{zero} \mathrel{/^?} i]\ \mathsf{zero} & \hookrightarrow i \end{cases} \qquad P_3 = \begin{cases} [\mathsf{suc}\ p \mathrel{/^?} i]\ \mathsf{zero} & \hookrightarrow i \\ [p \mathrel{/^?} k] \quad (\mathsf{suc}\ l) & \hookrightarrow \mathsf{suc}\ (\mathsf{max}\ k\ l) \end{cases} \tag{8}$$

We continue applying these operations Intro and 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 user-written pattern on the left is a constructor. For example, for P_2, we get after one more introduction step $(n : \mathbb{N}) \vdash P_4 \mid \mathsf{max}\ \mathsf{zero}\ n : \mathbb{N}$ where:

$$P_4 = \begin{cases} [n \mathrel{/^?} j] & \hookrightarrow j \\ [\mathsf{zero} \mathrel{/^?} i, n \mathrel{/^?} \mathsf{zero}] & \hookrightarrow i \end{cases} \tag{9}$$

We solve the remaining constraint in the first clause by instantiating $j := n$. This means we are done and we have $\mathsf{max}\ \mathsf{zero}\ n = j[n \mathrel{/} j] = n$ (rule Done).[8] The second clause of P_4 still has unsolved constraints, but this is not a problem since it is not used for the construction of this branch of the case tree.

Similarly, elaborating the lhs problem $(p : \mathbb{N}) \vdash P_3 \mid \mathsf{max}\ (\mathsf{suc}\ p) : \mathbb{N} \to \mathbb{N}$ (with rules Intro, SplitCon, and Done) gives us $\mathsf{max}\ (\mathsf{suc}\ p)\ \mathsf{zero} = \mathsf{suc}\ p$ and $\mathsf{max}\ (\mathsf{suc}\ p)\ (\mathsf{suc}\ q) = \mathsf{suc}\ (\mathsf{max}\ p\ q)$.

We record the operations used when elaborating the clauses in a *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 $\mathsf{case}_x\{\}$ for a case split. For max, we get the following case tree:

$$\lambda m.\ \mathsf{case}_m \left\{ \begin{array}{l} \mathsf{zero} \mapsto \lambda n.\ n \\ \mathsf{suc}\ p \mapsto \lambda n.\ \mathsf{case}_n \left\{ \begin{array}{l} \mathsf{zero} \mapsto \mathsf{suc}\ p \\ \mathsf{suc}\ q \mapsto \mathsf{suc}\ (\mathsf{max}\ p\ q) \end{array} \right\} \end{array} \right\} \tag{10}$$

[7] At this point, we could also introduce the variable for the second argument of max, the elaboration algorithm is free to choose either option.

[8] Since the Done rule only looks at the *first* remaining clause, there may be unreachable clauses that are not used in any of the branches of the case tree. However, detection of unreachable clauses can easily be added as an additional check if desired.

2.2 Copattern matching

Next, we take a look at how to elaborate definitions using copatterns. For the cozero example (1), we have the initial lhs problem $\vdash P_0 \mid \mathsf{cozero} : \mathsf{CoNat}$ where:

$$P_0 = \begin{cases} .\mathsf{iszero} & \hookrightarrow \mathsf{true} \\ .\mathsf{pred}\ \emptyset & \hookrightarrow \mathsf{impossible} \end{cases} \tag{11}$$

Here, we need a new operation to split on the result type CoNat (rule CoSplit). This produces two subproblems $\vdash P_1 \mid \mathsf{cozero}\ .\mathsf{iszero}$ and $\vdash P_2 \mid \mathsf{cozero}\ .\mathsf{pred} : \mathsf{cozero}\ .\mathsf{iszero} \equiv_{\mathsf{Bool}} \mathsf{false} \to \mathsf{CoNat}$ where:

$$P_1 = \{ \hookrightarrow \mathsf{true} \qquad P_2 = \{ \emptyset \hookrightarrow \mathsf{impossible} \tag{12}$$

The first problem is solved immediately with $\mathsf{cozero}\ .\mathsf{iszero} = \mathsf{true}$ (rule Done). In the second problem, we introduce the variable $x : \mathsf{cozero}\ .\mathsf{iszero} \equiv_{\mathsf{Bool}} \mathsf{false}$ (rule Intro) and note that $\mathsf{cozero}\ .\mathsf{iszero} = \mathsf{true}$ from the previous branch, hence $x : \mathsf{true} \equiv_{\mathsf{Bool}} \mathsf{false}$. Since $\mathsf{true} \equiv_{\mathsf{Bool}} \mathsf{false}$ is an empty type (technically, since unification of true with false results in a conflict), we can perform a case split on x with zero cases (rule SplitEmpty), solving the problem.

In the resulting case tree, the syntax for a split on the result type is record{}, with one subtree for each field of the record type:

$$\mathsf{record} \left\{ \begin{array}{ll} \mathsf{iszero} & \mapsto \mathsf{true} \\ \mathsf{pred} & \mapsto \lambda x.\ \mathsf{case}_x\{\} \end{array} \right\} \tag{13}$$

Just as pattern matching is elaborated to a case, an elimination form, copattern matching is elaborated to a record, an introduction form.

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.

2.3 Mixing patterns and copatterns

Consider the type CStream of C streams: potentially infinite streams of numbers that end on a zero. We define this as a record where the tail field has two extra arguments enforcing that we can only take the tail if the head is suc *m* for some *m*:

$$\begin{array}{l} \mathsf{record}\ \mathit{self} : \mathsf{CStream} : \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{head} : \mathbb{N} \\ \quad \mathsf{tail}\ \ : (m : \mathbb{N}) \to \mathit{self}\ .\mathsf{head} \equiv_{\mathbb{N}} \mathsf{suc}\ m \to \mathsf{CStream} \end{array} \tag{14}$$

Here, the name *self* is bound to the current record instance, allowing later projections to depend on prior projections. In particular, the type of tail depends on the value of head for the *self* object.

Now consider the function countdown that creates a C stream counting down from a given number n:

$$\begin{aligned}
&\mathsf{countdown} : \mathbb{N} \to \mathsf{CStream} \\
&\mathsf{countdown}\ n\ .\mathsf{head} = n \\
&\mathsf{countdown}\ \mathsf{zero}\ .\mathsf{tail}\ m\ \emptyset \\
&\mathsf{countdown}\ (\mathsf{suc}\ m)\ .\mathsf{tail}\ m\ \mathsf{refl} = \mathsf{countdown}\ m
\end{aligned} \tag{15}$$

Because the type of tail depends on the value of head, it is again important to check the clauses in the right order: to check the pattern $\emptyset$ in the second clause, the typechecker needs to know that countdown zero .head = zero (and likewise that countdown (suc m) .head = suc m to check refl in the third clause).

Our elaboration algorithm applies the rules Intro, Cosplit, SplitCon, SplitEmpty, SplitEq, and Done in sequence to translate this definition to the following case tree:

$$\lambda n.\ \mathsf{record}\left\{\begin{array}{l}\mathsf{head} \mapsto n \\ \mathsf{tail} \mapsto \lambda m, p.\ \mathsf{case}_n \left\{\begin{array}{l}\mathsf{zero} \mapsto \mathsf{case}_p\{\} \\ \mathsf{suc}\ n' \mapsto \mathsf{case}_p\ \{\mathsf{refl} \mapsto^{\mathbb{1}_m} (\mathsf{countdown}\ m)\}\end{array}\right.\end{array}\right\} \tag{16}$$

Note the extra annotation $\mathbb{1}_m$ after the case split on $p : \mathsf{suc}\ m \equiv_{\mathbb{N}} \mathsf{suc}\ n'$. This rather technical addition is necessary to define the operational semantics of case trees (see Figure 8). It is used to remember which arguments correspond to forced patterns and can thus safely be dropped. In the current example, it reflects the fact that the argument n' is forced to be equal to m by the case split on $\mathsf{refl} : \mathsf{suc}\ n' \equiv_{\mathbb{N}} \mathsf{suc}\ m$. Thus, evaluation of the function only depends on the value of m.

2.4 Inferring forced patterns

This example is based on issue #2896 on the Agda bug tracker (Agda issue, 2018a). 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 data type D and function foo:

$$\begin{aligned}
&\mathsf{data}\ \mathsf{D}\ (m : \mathbb{N}) : \mathsf{Set}\ \mathsf{where} \\
&\quad \mathsf{c} : (n : \mathbb{N})\ (p : n \equiv_{\mathbb{N}} m) \to \mathsf{D}\ m
\end{aligned} \qquad
\begin{aligned}
&\mathsf{foo} : (m : \mathbb{N}) \to \mathsf{D}\ (\mathsf{suc}\ m) \to \mathbb{N} \\
&\mathsf{foo}\ m\ (\mathsf{c}\ (\mathsf{suc}\ k)\ \mathsf{refl}) = m + k
\end{aligned} \tag{17}$$

The old algorithm would ignore the pattern suc k in the definition of foo because it corresponds to a forced pattern after the case split on refl. This lead to the variable k occurring in the case tree despite it not being bound anywhere. Our elaboration instead produces the following case tree (using rules Intro, SplitCon, SplitEq, and Done):

$$\lambda m, x.\ \mathsf{case}_x\{\ \mathsf{c}\ n\ p \mapsto\ \mathsf{case}_p\ \{\mathsf{refl} \mapsto^{\mathbb{1}_m} (m + m)\}\ \} \tag{18}$$

Note that the variable k has been substituted by m in the leaf of the case tree. Even though this case tree does not match on the suc constructor, it implements the same computational behaviour as the clause in the definition of foo because the first argument of c is *forced* to be suc m by the typing rules.

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

2.5 Preservation of first-match semantics

This example is based on issue #2964 on the Agda bug tracker (Agda issue, 2018b). 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:

$$\begin{aligned} &\mathsf{f} : (A : \mathsf{Set}) \to A \to \mathsf{Bool} \to (A \equiv_{\mathsf{Set}} \mathsf{Bool}) \to \mathsf{Bool} \\ &\mathsf{f}\ \lfloor \mathsf{Bool} \rfloor\ \mathsf{true}\ \mathsf{true}\ \mathsf{refl} = \mathsf{true} \\ &\mathsf{f}\ \_\ \_\ \_\ \_ \qquad\qquad\quad = \mathsf{false} \end{aligned} \tag{19}$$

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

$$\lambda A, x, y, p.\ \mathsf{case}_y \left\{ \begin{array}{l} \mathsf{true} \mapsto \mathsf{case}_p \left\{ \mathsf{refl} \mapsto^{\mathbb{1}_{x,y}} \mathsf{case}_x \left\{ \begin{array}{l} \mathsf{true} \mapsto \mathsf{true} \\ \mathsf{false} \mapsto \mathsf{false} \end{array} \right\} \right\} \\ \mathsf{false} \mapsto \mathsf{false} \end{array} \right\} \tag{20}$$

According to the (liberal) first-match semantics, we should have $\mathsf{f}\ \mathsf{Bool}\ \mathsf{false}\ y\ p = \mathsf{false}$ for any $y : \mathsf{Bool}$ and $p : \mathsf{Bool} \equiv_{\mathsf{Set}} \mathsf{Bool}$, but this is not true for the case tree since evaluation gets stuck on the variable y. Another possibility is to start the case tree by a split on p (after introducing all the variables), but this case tree still gets stuck on the variable 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 $x : A$ before splitting on 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 f in this example, because the type of x is not known to be a data type until the case split on $A \equiv_{\mathsf{Set}} \mathsf{Bool}$. Instead, we choose to keep the elaboration as it is and restrict the first-match semantics of clauses (see Sections 3.4 and 5.3). In particular, we use a more conservative matching algorithm that only produces a mismatch when not only at least one pattern is a mismatch (as usual), but also matching is not stuck for any of the other patterns. In the example of f, this change means that we can only go to the second clause once all three arguments x, y, and p are constructors, and at least one of them produces a mismatch. In particular, this means $\mathsf{f}\ \mathsf{Bool}\ \mathsf{false}\ y\ p$ no longer computes to false. This is reasonable since there exists a valid case tree that does not have this computation rule.

2.6 Expanding catch-all clauses

Compared to the implementation in Agda, the algorithm for elaborating definitions by dependent (co)pattern matching to a case tree in this paper is more liberal with respect to *catch-all cases*. Specifically, Agda includes an extra pass where each clause is type-checked *individually* before elaborating the whole set of clauses to a case tree. In contrast, the algorithm presented here only requires that the type of each right-hand side is correct after all case splitting is finished.

Let us consider an example where this makes a difference. Let Bin be the data type of binary numbers 1, 10, 11, 100, 101, ... and consider the function isOne:

$$\begin{aligned}
&\mathsf{data}\ \mathsf{Bin} : \mathsf{Set}\ \mathsf{where} \\
&\quad \mathsf{one}\ \ : \mathsf{Bin} \\
&\quad \mathsf{2*}\ \ : \mathsf{Bin} \to \mathsf{Bin} \\
&\quad \mathsf{1+2*} : \mathsf{Bin} \to \mathsf{Bin}
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{isOne} : \mathsf{Bin} \to \mathsf{Bool} \\
&\mathsf{isOne}\ \mathsf{one} = \mathsf{true} \\
&\mathsf{isOne}\ x\ \ = \mathsf{false}
\end{aligned}
\tag{21}$$

For example, the number 9 is represented as 1+2* (2* (2* one)). Now consider the following soundness proof isOneSound below:

$$\begin{aligned}
&\mathsf{isOneSound} : (x : \mathsf{Bin}) \to \mathsf{isOne}\ x \equiv_{\mathsf{Bool}} \mathsf{true} \to x \equiv_{\mathsf{Bin}} \mathsf{one} \\
&\mathsf{isOneSound}\ \mathsf{one}\ \mathsf{refl} = \mathsf{refl} \\
&\mathsf{isOneSound}\ x\ \emptyset
\end{aligned}
\tag{22}$$

Following the algorithm of Section 5.2, this definition is elaborated to the following case tree:

$$\lambda x.\ \lambda p.\ \mathsf{case}_x \left\{ \begin{array}{lll} \mathsf{one} & \mapsto & \mathsf{case}_p \{ \mathsf{refl} \mapsto \mathsf{refl} \} \\ \mathsf{2*} & \mapsto & \mathsf{case}_p \{\} \\ \mathsf{1+2*} & \mapsto & \mathsf{case}_p \{\} \end{array} \right\} \tag{23}$$

On the other hand, this definition is rejected by Agda because the second clause is ill-typed: x is not a constructor so the term isOne x does not reduce to either true or false, hence Agda cannot conclude that isOne $x \equiv_{\mathsf{Bool}}$ true is an empty type.

One advantage of the approach in this paper is that it allows for shorter definitions of some functions. For example, the shortest possible definition of isOneSound in Agda requires three clauses. As another example, proving decidable equality for a data type D with n constructors in general requires n^2 cases in Agda, while it only requires $n + 1$ cases with the elaboration algorithm presented here:

$$\begin{aligned}
&\mathsf{decEq} : (x\ y : D) \to \mathsf{Dec}\ (x \equiv_D y) \\
&\mathsf{decEq}\ (\mathsf{c}_1\ \bar{x})\ (\mathsf{c}_1\ \bar{y}) = \cdots \\
&\qquad\qquad \vdots \\
&\mathsf{decEq}\ (\mathsf{c}_n\ \bar{x})\ (\mathsf{c}_n\ \bar{y}) = \cdots \\
&\mathsf{decEq}\ _\ _\qquad\quad = \mathsf{no}\ (\lambda\ \emptyset)
\end{aligned}
\tag{24}$$

here, the final clause is expanded into $n \cdot (n-1)$ leaves of the case tree corresponding to the cases where the two arguments are applications of different constructors.

However, it is not clear how this late checking of the right-hand side should interact with other Agda features beyond the scope of this paper, such as *interaction holes* used for interactively developing Agda programmes. If such an interaction hole occurs in the right-hand side of a clause that corresponds to multiple branches of the case tree, then it may have multiple incompatible types. At this point, it is unclear how to display this information in a way that makes sense to the Agda user. For this reason, we have decided to leave the initial pass over the individual clauses in place for now.

The behaviour of Agda can be simulated by the elaboration judgement described in this paper by additionally requiring that each clause individually can be elaborated to a case tree. These individual case trees can then be 'merged' into one bigger case tree by following the first-match semantics. Alternatively, we can run the elaboration algorithm again on the whole set of clauses (as is currently done in the Agda implementation).

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 data types, 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 data types (other than the equality type). These features can nevertheless be encoded in our language, see Section 3.5 for details.

Note also that we do not include any rules for η-equality, neither for lambda expressions (which do not exist) nor for records (which can be coinductive hence do not satisfy η). Section 3.5 discusses how our language could be extended with η-rules.

3.1 Syntax of the core type theory

Expression syntax. 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 symbol f. We generalize applications to eliminations e by including projections $.\pi$ in spines $\bar{e}$. Any expression is in weak head normal form but $\mathsf{f}\,\bar{e}$, which is computed via pattern matching (see Section 3.4):

$$
\begin{array}{lll}
A, B, u, v & ::= w & \text{weak head normal form} \\
& \mid \mathsf{f}\,\bar{e} & \text{defined symbol applied to eliminations} \\
W, w & ::= (x : A) \to B & \text{dependent function type} \\
& \mid \mathsf{Set}_\ell & \text{universe } \ell \\
& \mid \mathsf{D}\,\bar{u} & \text{data type fully applied to parameters} \\
& \mid \mathsf{R}\,\bar{u} & \text{record type fully applied to parameters} \\
& \mid u \equiv_A v & \text{equality type} \\
& \mid x\,\bar{e} & \text{variable applied to eliminations} \\
& \mid \mathsf{c}\,\bar{u} & \text{constructor fully applied to arguments} \\
& \mid \mathsf{refl} & \text{proof of reflexivity}
\end{array}
\tag{25}
$$

Any expression but $\mathsf{c}\,\bar{u}$ or 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 Set_ℓ. Syntax is coloured according to the Agda conventions: primitives and defined symbols are blue, constructors are green, and projections are pink:

$$\begin{array}{rll} e ::= & u & \text{application} \\ \mid & .\pi & \text{projection} \end{array} \tag{26}$$

Binary application $\boxed{u\,e}$ 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 $(\mathsf{f}\,\bar{e})\,e = \mathsf{f}\,(\bar{e}, e)$, respectively, otherwise it is undefined. The expression syntax does not include anonymous functions or record expressions, but these can be defined in terms of definitions by (co)pattern matching (see Section 3.5).

Pattern syntax. 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*:

$$\begin{array}{rlr} p ::= & x & \text{variable pattern} \\ \mid & \mathsf{refl} & \text{pattern for reflexivity proof} \\ \mid & \mathsf{c}\,\bar{p} & \text{constructor pattern} \\ \mid & \lfloor \mathsf{c} \rfloor\,\bar{p} & \text{forced constructor pattern} \\ \mid & \lfloor u \rfloor & \text{forced argument} \\ \mid & \emptyset & \text{absurd pattern} \end{array} \tag{27}$$

$$\begin{array}{rlr} q ::= & p & \text{application copattern} \\ \mid & .\pi & \text{projection copattern} \end{array}$$

Forced patterns (Brady *et al.*, 2003) appear with dependent types; they are either entirely forced arguments $\lfloor u \rfloor$, which are Agda's *dot patterns*, or only the constructor is forced $\lfloor \mathsf{c} \rfloor\,\bar{p}$. An argument can be forced by a match against 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[9] 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 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 $\mathsf{tt}\,\emptyset = \mathsf{impossible}$.

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

$$\begin{array}{lll} \lceil x \rceil = x & \lceil \mathsf{c}\,\bar{p} \rceil = \mathsf{c}\,\lceil \bar{p} \rceil & \lceil \lfloor u \rfloor \rceil = u \\ \lceil \mathsf{refl} \rceil = \mathsf{refl} & \lceil \lfloor \mathsf{c} \rfloor\,\bar{p} \rceil = \mathsf{c}\,\lceil \bar{p} \rceil & \lceil .\pi \rceil = .\pi \end{array} \tag{28}$$

Telescopes and contexts. Constructors take a list of arguments whose types can depend on all previous arguments. The constructor parameters are given as a list $x_1{:}A_1, \ldots, x_n{:}A_n$

[9] Absurd patterns are written as '()' in Agda syntax.

$$
\begin{array}{lcll}
s & ::= & \ominus & \text{status: unchecked} \\
 & | & \oplus & \text{status: checked} \\
\\
\mathit{decl}^{s} & ::= & \mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_{\ell}\ \mathsf{where}\ \overline{\mathit{con}} & \text{datatype declaration} \\
 & | & \mathsf{record}\ \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_{\ell}\ \mathsf{where}\ \overline{\mathit{field}} & \text{record declaration} \\
 & | & \mathsf{definition}\ \mathsf{f} : A\ \mathsf{where}\ \overline{\mathit{cls}^{s}} & \text{function declaration} \\
\\
\mathit{con} & ::= & \mathsf{c}\ \Delta & \text{constructor declaration} \\
\\
\mathit{field} & ::= & \pi : A & \text{field declaration} \\
\\
\mathit{cls}^{\ominus} & ::= & \bar{q} \hookrightarrow \mathit{rhs} & \text{unchecked clause} \\
\mathit{cls}^{\oplus} & ::= & \Delta \vdash \bar{q} \hookrightarrow u : B & \text{checked clause} \\
\\
\mathit{rhs} & ::= & u & \text{clause body: expression} \\
 & | & \mathsf{impossible} & \text{empty body for absurd pattern} \\
\\
\Sigma & ::= & \overline{\mathit{decl}^{\oplus}} & \text{signature}
\end{array}
$$

Fig. 1. Declarations.

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* (de Bruijn, 1991), or as a *snoc*-list, then we call it a *context*:

$$
\begin{array}{lll}
\Gamma ::= \varepsilon & & \text{empty context} \\
\quad | \quad \Gamma(x : A) & & \text{context extension} \\
\Delta ::= \varepsilon & & \text{empty telescope} \\
\quad | \quad (x : A)\Delta & & \text{non-empty telescope}
\end{array}
\tag{29}
$$

Context and telescopes can be regarded as finite maps from variables to types, and we require $x \notin \mathrm{dom}(\Gamma)$ and $x \notin \mathrm{dom}(\Delta)$ 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 Γ is a context and Δ a telescope in context Γ where $\mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta)$ are disjoint, then $\boxed{\Gamma\,\Delta}$ defined by $\Gamma\,\varepsilon = \Gamma$ 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 Δ, let $\boxed{\hat{\Delta}}$ be Δ without the types, i.e. the variables of Δ in order. Similarly, we write $\hat{\Gamma}$ for the variables bound in the context Γ. Further, we define $\boxed{\Delta \to C}$ as the iterated dependent function type via $\varepsilon \to C = C$ and $(x{:}A)\Delta \to C = (x{:}A) \to (\Delta \to C)$.

Declaration syntax. 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 (see Figure 1). The input to the type checker is a list of unchecked declarations $\mathit{decl}^{\ominus}$, and the output a list of checked declarations $\mathit{decl}^{\oplus}$, called a *signature* Σ.

A *data type* D can be parameterized by telescope Δ and inhabits one of the universes Set_{ℓ}. Each of its constructors c_i (although there might be none) takes a telescope Δ_i of arguments that can refer to the parameters in Δ. The full type of c_i could be $\Delta\Delta_i \to D\hat{\Delta}$, but we never apply constructors to the data parameters explicitly.

A *record type* R can be thought of as data type with a single constructor; 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 $self\ .\pi_i$ where variable *self* is a placeholder for the value of the whole record.[10] The full type of projection π_i could be $\Delta(self : \mathsf{R}\hat{\Delta}) \to A_i$, 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 both function and record definitions. Each clause defining the constant $\mathsf{f} : A$ consists of a list of copatterns $\bar{q}$ (including patterns and projections) and right-hand side *rhs*. The copatterns eliminate type A into the type of the *rhs* which is either a term u or the special keyword impossible, in case one of the copatterns q_i contains an absurd pattern $\emptyset$. The intended semantics is that if an application $\mathsf{f}\,\bar{e}$ matches a left-hand side $\mathsf{f}\,\bar{q}$ with substitution σ, then $\mathsf{f}\,\bar{e}$ reduces to *rhs* under σ. For efficient computation of matching, we require *linearity of pattern variables* for checked clauses: each variable in $\bar{q}$ occurs only once in a non-forced position. Note that checked/unchecked declarations can only mention checked/unchecked clauses, respectively.

While checking declarations, the typechecker builds up a signature Σ of already checked declarations or parts of declarations. Checked clauses are the elaboration (Sections 2 and 5) of the corresponding unchecked clauses: they are non-overlapping and supplemented by a telescope Δ 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:

$$
\begin{array}{rll}
Z ::= & \mathsf{data}\ D\ \Delta : \mathsf{Set}_\ell & \text{data type signature} \\
\mid & \mathsf{constructor}\ \mathsf{c}\ \Delta_\mathsf{c} : \mathsf{D}\ \Delta & \text{constructor signature} \\
\mid & \mathsf{record}\ \mathsf{R}\ \Delta : \mathsf{Set}_\ell & \text{record type signature} \\
\mid & \mathsf{projection}\ self : \mathsf{R}\ \Delta \vdash .\pi : A & \text{projection signature} \\
\mid & \mathsf{definition}\ \mathsf{f} : A & \text{function signature} \\
\mid & \mathsf{clause}\ \Delta \vdash \mathsf{f}\,\bar{q} \hookrightarrow v : B & \text{function clause}
\end{array}
\tag{30}
$$

Adding a snippet Z to a signature Σ, written $\boxed{\Sigma, Z}$ is a always defined if Z is a data or record type or function signature; in this case, the corresponding declaration is appended to Σ. Adding a constructor signature $\mathsf{constructor}\ \mathsf{c}\ \Delta_\mathsf{c} : \mathsf{D}\ \Delta$ is only defined if the *last* declaration in Σ is $(\mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \overline{con})$ and c is not part of $\overline{con}$ yet. Analogous conditions apply when adding projection snippets. Function clauses can be added if the

[10] *self* is the analogous of Java's 'this', but like in Scala's 'trait', the name can be chosen.

Table 1. *List of typing and equality judgements of our core type theory.*

Judgement	Explanation	Rules
$\Sigma \vdash \Gamma$	Context Γ is well formed.	Figure A.1
$\Sigma; \Gamma \vdash_\ell \Delta$	In Γ, telescope Δ is well formed and bounded by level ℓ.	Figure 4
$\Sigma; \Gamma \vdash u : A$	In Γ, term u has type A.	Figure 2
$\Sigma; \Gamma \vdash \bar{u} : \Delta$	In Γ, term list $\bar{u}$ instantiates telescope Δ.	Figure 4
$\Sigma; \Gamma \mid u : A \vdash \bar{e} : B$	In Γ, head u of type A is eliminated via $\bar{e}$ to type B.	Figure 3
$\Sigma; \Gamma \vdash u = v : A$	In Γ, terms u and v are equal of type A.	Figure A.2
$\Sigma; \Gamma \vdash \bar{u} = \bar{v} : \Delta$	In Γ, term lists $\bar{u}$ and $\bar{v}$ are equal instantiations of Δ.	Figure A.4
$\Sigma; \Gamma \mid u : A \vdash \bar{e} = \bar{e}' : B$	In Γ, $\bar{e}$ and $\bar{e}'$ are equal eliminations of head $u : A$ to type B.	Figure A.3

last declaration of Σ is a function declaration with the same name. We trust the formal definition of Σ, 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 for functions that have already been checked. Late addition of function clauses would not pose a problem, but since we require all function definitions to be total, any additional clauses would anyway be unreachable.

Membership of a snippet is written $\boxed{Z \in \Sigma}$ 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 basic typing and equality judgements listed in Table 1. In all these judgements, the signature Σ 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 ℓ:

$$\begin{aligned}
\Sigma; \Gamma \vdash \Delta &\iff \exists \ell.\ \Sigma; \Gamma \vdash_\ell \Delta && \text{well-formed telescope} \\
\Sigma; \Gamma \vdash A &\iff \exists \ell.\ \Sigma; \Gamma \vdash A : \mathsf{Set}_\ell && \text{well-formed type} \\
\Sigma; \Gamma \vdash A = B &\iff \exists \ell.\ \Sigma; \Gamma \vdash A = B : \mathsf{Set}_\ell && \text{equal types}
\end{aligned}$$

In the inference rules, we make use of substitutions. *Substitutions* σ, τ, ν are partial maps from variable names to terms with a finite domain. If $\mathrm{dom}(\sigma)$ and $\mathrm{dom}(\tau)$ are disjoint, then $\boxed{\sigma \uplus \tau}$ 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 $\bar{v} = v_1, \ldots, v_n$ is a list of terms, we may write $[\bar{v} / \Delta]$ for the substitution $[\bar{v} / \hat{\Delta}]$, i.e. $[v_1 / x_1; \ldots; v_n / x_n]$. In particular, the identity substitution $\boxed{\mathbb{1}_\Gamma} := [\hat{\Gamma} / \Gamma]$ maps all variables in Γ to themselves. We also use the identity substitution as a weakening substitution, allowing us to forget about all variables that are not in Γ. If $x \in \mathrm{dom}(\sigma)$, then $\boxed{\sigma \setminus x}$ is defined by removing x from the domain of σ.

Application of a substitution σ to a term u is written as $\boxed{u\sigma}$ and is defined as usual by replacing all (free) variables in u by their values given by σ, avoiding variable capture via

$\boxed{\Gamma \vdash u : A}$ Entails $\vdash \Gamma$ and $\Gamma \vdash A$.

Types.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Set}_\ell : \mathsf{Set}_{\ell+1}} \qquad \frac{\Gamma \vdash A : \mathsf{Set}_\ell \qquad \Gamma (x : A) \vdash B : \mathsf{Set}_{\ell'}}{\Gamma \vdash (x : A) \to B : \mathsf{Set}_{\mathrm{max}(\ell,\ell')}}$$

$$\frac{\mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell \in \Sigma \qquad \Gamma \vdash \bar{u} : \Delta}{\Gamma \vdash \mathsf{D}\ \bar{u} : \mathsf{Set}_\ell} \qquad \frac{\mathsf{record}\ \mathsf{R}\ \Delta : \mathsf{Set}_\ell \in \Sigma \qquad \Gamma \vdash \bar{u} : \Delta}{\Gamma \vdash \mathsf{R}\ \bar{u} : \mathsf{Set}_\ell}$$

$$\frac{\Gamma \vdash A : \mathsf{Set}_\ell \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A}{\Gamma \vdash u \equiv_A v : \mathsf{Set}_\ell}$$

Heads ($h ::= x\ \varepsilon \mid \mathsf{f}\ \varepsilon$) and applications $h\ \bar{e}$.

$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x\ \varepsilon : A} \qquad \frac{\vdash \Gamma \qquad \mathsf{definition}\ \mathsf{f} : A \in \Sigma}{\Gamma \vdash \mathsf{f}\ \varepsilon : A} \qquad \frac{\Gamma \vdash h : A \qquad \Gamma \mid h : A \vdash \bar{e} : C}{\Gamma \vdash h\ \bar{e} : C}$$

Values.

$$\frac{\mathsf{constructor}\ \mathsf{c}\ \Delta_\mathsf{c} : \mathsf{D}\ \Delta \in \Sigma \qquad \Gamma \vdash \bar{u} : \Delta \qquad \Gamma \vdash \bar{v} : \Delta_\mathsf{c}[\bar{u} / \Delta]}{\Gamma \vdash \mathsf{c}\ \bar{v} : \mathsf{D}\ \bar{u}}$$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash \mathsf{refl} : u \equiv_A u}$$

Conversion.

$$\frac{\Gamma \vdash u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash u : B}$$

Fig. 2. Typing rules for expressions.

suitable renaming of bound variables. Like function application, this is a partial operation on the syntax; for instance, $(x\ .\pi)[\mathsf{c}\ /\ x]$ 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 (established in Lemma 3). Substitution composition $\boxed{\sigma ; \tau}$ shall map the variable x to the term $(x\sigma)\tau$. Note the difference between $\sigma ; \tau$ and $\sigma \uplus \tau$: the former applies first σ and then τ in sequence, while the latter applies σ and τ in parallel to disjoint parts of the context. Application of a substitution to a pattern $\boxed{p\sigma}$ is defined as $\lceil p \rceil \sigma$.

In addition to substitutions on terms, we also make use of substitutions on patterns called *pattern substitutions*. A pattern substitution ρ assigns to each variable a pattern. We reuse the same syntax for pattern substitutions as for normal substitutions. Any pattern substitution ρ can be used as a normal substitution $\lceil \rho \rceil$ defined by $x \lceil \rho \rceil = \lceil x\rho \rceil$.

The rules for the typing judgement $\boxed{\Gamma \vdash t : A}$ are listed in Figure 2. The type formation rules introduce an infinite hierarchy of predicative universes Set_ℓ without cumulativity. The formation rules for data and record types make use of the judgement $\Gamma \vdash \bar{u} : \Delta$ to type argument lists, same for the constructor rule, which introduces a data type. Further, refl introduces the equality type. All expressions involved in these rules are fully applied,

$\boxed{\Gamma \mid u : A \vdash \bar{e} : C}$ If $\Gamma \vdash u : A$ then $\Gamma \vdash C$.

$$\frac{}{\Gamma \mid u : A \vdash \varepsilon : A} \qquad \frac{\Gamma \vdash v : A \qquad \Gamma \mid u\, v : B[v/x] \vdash \bar{e} : C}{\Gamma \mid u : (x : A) \to B \vdash v\, \bar{e} : C}$$

$$\frac{\mathsf{projection}\ \mathit{self} : \mathsf{R}\ \Delta \vdash .\pi : A \in \Sigma \qquad \Gamma \mid u\ .\pi : A[\bar{v}/\Delta, u/\mathit{self}] \vdash \bar{e} : C}{\Gamma \mid u : \mathsf{R}\ \bar{v} \vdash .\pi\ \bar{e} : C}$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma \mid u : A' \vdash \bar{e} : C}{\Gamma \mid u : A \vdash \bar{e} : C}$$

Fig. 3. The typing rules for eliminations.

$\boxed{\Gamma \vdash_\ell \Delta}$ Entails $\vdash \Gamma$.

$$\frac{\vdash \Gamma}{\Gamma \vdash_\ell \varepsilon} \qquad \frac{\Gamma \vdash A : \mathsf{Set}_{\ell'} \qquad \Gamma(x : A) \vdash_\ell \Delta}{\Gamma \vdash_\ell (x : A)\Delta}\ \ell' \leq \ell$$

$\boxed{\Gamma \vdash \bar{u} : \Delta}$ Precondition: $\Gamma \vdash \Delta$.

$$\frac{}{\Gamma \vdash \varepsilon : \varepsilon} \qquad \frac{\Gamma \vdash u : A \qquad \Gamma \vdash \bar{u} : \Delta[u/x]}{\Gamma \vdash u\, \bar{u} : (x : A)\Delta}$$

Fig. 4. The typing rules for telescopes and lists of terms.

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 $\bar{e}$, judgement $\boxed{\Gamma \mid u : A \vdash \bar{e} : C}$, are presented in Figure 3. For checking arguments, the type of the head is sufficient, and it needs to be a function type. To check projections, we also need 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 1 (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 (Section 3.4) and, equivalently, case trees (Section 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.

Figure 4 defines judgement $\boxed{\Gamma \vdash_\ell \Delta}$ for telescope formation. The level ℓ 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 Σ-type, then ℓ is an upper bound for the universe that hosts

$\boxed{\Sigma \vdash Z}$ Snippet Z is well-formed in signature Σ.

$$\frac{\Sigma \vdash \Delta}{\Sigma \vdash \mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell} \qquad \frac{\mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell \in \Sigma \qquad \Sigma; \Delta \vdash_\ell \Delta_\mathsf{c}}{\Sigma \vdash \mathsf{constructor}\ \mathsf{c}\ \Delta_\mathsf{c} : \mathsf{D}\ \Delta}$$

$$\frac{\Sigma \vdash \Delta}{\Sigma \vdash \mathsf{record}\ \mathsf{R}\ \Delta : \mathsf{Set}_\ell} \qquad \frac{\Sigma \vdash A}{\Sigma \vdash \mathsf{definition}\ \mathsf{f} : A}$$

$$\frac{\mathsf{record}\ \mathsf{R}\ \Delta : \mathsf{Set}_\ell \in \Sigma \qquad \Sigma; \Delta(x : \mathsf{R}\ \hat{\Delta}) \vdash A : \mathsf{Set}_{\ell'}}{\Sigma \vdash (\mathsf{projection}\ x : \mathsf{R}\ \Delta \vdash .\pi : A)}\ \ell' \leq \ell$$

$$\frac{\mathsf{definition}\ \mathsf{f} : A \in \Sigma \qquad \Sigma \vdash \Delta \qquad \Delta \mid \mathsf{f} : A \vdash \lceil \bar{q} \rceil : B \qquad \Delta \vdash v : B}{\Sigma \vdash \mathsf{clause}\ \Delta \vdash \mathsf{f}\ \bar{q} \hookrightarrow v : B}$$

$\boxed{\Sigma_0 \subseteq \Sigma}$ Signature Σ is a valid extension of Σ_0.

$$\frac{}{\Sigma_0 \subseteq \Sigma_0} \qquad \frac{\Sigma_0 \subseteq \Sigma \qquad \Sigma \vdash Z \qquad \Sigma, Z\ \text{defined}}{\Sigma_0 \subseteq \Sigma, Z}$$

Fig. 5. Rules for well-formed signature snippets and extension.

this type. This is important when checking that the level of a data type is sufficiently high for the level of data it contains (Figure 5).

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 $\boxed{\Gamma \vdash \sigma : \Delta}$ for $\mathrm{dom}(\sigma) = \Delta$ and $\Gamma \vdash \hat{\Delta}\sigma : \Delta$. Likewise, we write $\boxed{\Gamma \vdash \sigma = \sigma' : \Delta}$ for $\Gamma \vdash \hat{\Delta}\sigma = \hat{\Delta}\sigma' : \Delta$.

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

$$\frac{\mathsf{clause}\ \Delta \vdash \mathsf{f}\ \bar{q} \hookrightarrow v : B \in \Sigma \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash \mathsf{f}\ \bar{q}\sigma = v\sigma : B\sigma}$$

If $\mathsf{f}\ \bar{q} \hookrightarrow v$ is a defining clause of function f, then each instance arising from a well-typed substitution σ is a valid equation. The full list of congruence and equivalence rules is given in Figure A.2 in Appendix A., together with congruence rules for applications (Figure A.3) and lists of terms (Figure A.4). As usual in dependent type theory, definitional equality on types $\Gamma \vdash A = B : \mathsf{Set}_\ell$ is used for type conversion.

Lemma 2. *If* $\Gamma \vdash \sigma : \Delta_1(x : A)\Delta_2$ *then also* $\Gamma \vdash \sigma : \Delta_1(\Delta_2[x\sigma\ /\ x])$.

Lemma 3 (Substitution)**.** *Suppose* $\Gamma' \vdash \sigma : \Gamma$. *Then the following hold*

- *If* $\Gamma \vdash u : A$ *then* $\Gamma' \vdash u\sigma : A\sigma$.
- *If* $\Gamma \mid u : A \vdash \bar{e} : B$ *then* $\Gamma' \mid u\sigma : A\sigma \vdash \bar{e}\sigma : B\sigma$.
- *If* $\Gamma \vdash_\ell \Delta$ *then* $\Gamma' \vdash_\ell \Delta\sigma$.
- *If* $\Gamma \vdash \bar{u} : \Delta$ *then* $\Gamma' \vdash \bar{u}\sigma : \Delta\sigma$.
- *If* $\Gamma \vdash u = v : A$ *then* $\Gamma' \vdash u\sigma = v\sigma : A\sigma$.
- *If* $\Gamma \mid u : A \vdash \bar{e}_1 = \bar{e}_2 : C$ *then* $\Gamma' \mid u\sigma : A\sigma \vdash \bar{e}_1\sigma = \bar{e}_2\sigma : C\sigma$.
- *If* $\Gamma \vdash \bar{u}_1 = \bar{u}_2 : \Delta$ *then* $\Gamma' \vdash \bar{u}_1\sigma = \bar{u}_2\sigma : \Delta$.

Proof By mutual induction on the derivation of the given judgement. The interesting case is when u is a variable application $x\,\bar{e}$. Suppose that $x : A \in \Gamma$ and $\Gamma \mid x : A \vdash \bar{e} : B$, then $\Gamma' \vdash x\sigma : A\sigma$. We also know from the induction hypothesis that $\Gamma' \mid x\sigma : A\sigma \vdash \bar{e}\sigma : B\sigma$, so we have $\Gamma' \vdash x\sigma\ \bar{e}\sigma : B\sigma$, as we had to prove. □

Property 4. *If* $\Gamma \vdash u : A$ *and* $\Gamma \mid u : A \vdash \bar{e} : B$*, then* $u\,\bar{e}$ *is well defined and* $\Gamma \vdash u\,\bar{e} : B$.

3.3 Signature well-formedness

A signature Σ *extends* Σ_0 if we can go from Σ_0 to Σ by adding valid snippets Z, i.e. new data types, record types, and defined constants, but new constructors/projections/clauses only for not yet completed definitions in Σ. A signature Σ is well formed if it is a valid extension of the empty signature ε. Formally, we define signature extension $\boxed{\Sigma_0 \subseteq \Sigma}$ via snippet typing $\boxed{\Sigma \vdash Z}$ by the rules in Figure 5, and signature well-formedness $\boxed{\vdash \Sigma}$ as $\varepsilon \subseteq \Sigma$. 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 ℓ; this preserves predicativity. However, the *parameters* Δ of a data or record type of level ℓ can be *big*, they may exceed ℓ.

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

Lemma 5 (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 6 (Progress and preservation)**.** 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 Section 4.

3.4 Pattern matching and evaluation rules

We define a small-step evaluation relation for our core language. This relation is not used by the typing judgement, but it serves as a reference point when proving the correctness of the operational semantics of case trees (Lemma 12). Since our language does not contain syntax for lambda abstraction, there is no rule for β-reduction. Almost all terms are their own weak head normal form; the only exception are applications $\mathsf{f}\,\bar{e}$. Formally, small-step evaluation $\boxed{\Sigma \vdash u \longrightarrow v}$ is defined as the congruence closure of the following rule:

$$\frac{\mathsf{clause}\ \Delta \vdash \mathsf{f}\,\bar{q} \hookrightarrow v : A \in \Sigma \qquad [\bar{e} \,/\, \bar{q}] \Rightarrow \sigma}{\Sigma \vdash \mathsf{f}\,\bar{e} \longrightarrow v\sigma} \tag{31}$$

The small-step semantics are related to the definitional equality judgement by the notion of *respectfulness* (see Definition 13).

$$\boxed{[v\,/\,p] \Rightarrow \sigma_\bot}$$

$$\frac{}{[v\,/\,x] \Rightarrow [v\,/\,x]} \qquad \frac{}{[v\,/\,\lfloor u \rfloor] \Rightarrow [\,]} \qquad \frac{}{[\mathsf{refl}\,/\,\mathsf{refl}] \Rightarrow [\,]}$$

$$\frac{[\bar{u}\,/\,\bar{p}] \Rightarrow \sigma_\bot}{[\mathsf{c}\,\bar{u}\,/\,\mathsf{c}\,\bar{p}] \Rightarrow \sigma_\bot} \qquad \frac{[\bar{u}\,/\,\bar{p}] \Rightarrow \sigma_\bot}{[\mathsf{c}_2\,\bar{u}\,/\,\lfloor \mathsf{c}_1 \rfloor\,\bar{p}] \Rightarrow \sigma_\bot} \qquad \frac{\mathsf{c}_1 \neq \mathsf{c}_2}{[\mathsf{c}_2\,\bar{u}\,/\,\mathsf{c}_1\,\bar{p}] \Rightarrow \bot}$$

$$\boxed{[e\,/\,q] \Rightarrow \sigma_\bot}$$

$$\frac{}{[.\pi\,/\,.\pi] \Rightarrow [\,]} \qquad \frac{\pi_1 \neq \pi_2}{[.\pi_2\,/\,.\pi_1] \Rightarrow \bot}$$

$$\boxed{[\bar{e}\,/\,\bar{q}] \Rightarrow \sigma_\bot}$$

$$\frac{}{[\varepsilon\,/\,\varepsilon] \Rightarrow [\,]} \qquad \frac{[e\,/\,q] \Rightarrow \sigma_\bot \qquad [\bar{e}\,/\,\bar{q}] \Rightarrow \tau_\bot}{[e\,\bar{e}\,/\,q\,\bar{q}] \Rightarrow \sigma_\bot \uplus \tau_\bot}$$

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

Evaluation of defined symbols relies on matching $\boxed{[\bar{e}\,/\,\bar{q}] \Rightarrow \sigma_\bot}$ (Figure 6). Herein, $\sigma_\bot$ is either a substitution σ with $\mathrm{dom}(\sigma) = \mathrm{PV}(\bar{q})$ or the error value $\bot$ for mismatch. Join of lifted substitutions $\sigma_\bot \uplus \tau_\bot$ is $\bot$ if one of the operands is $\bot$, otherwise the join $\sigma \uplus \tau$.

A pattern variable x matches any term v, producing singleton substitution $[v\,/\,x]$. Likewise for a forced pattern $\lfloor u \rfloor$, but it does not bind any pattern variables. Projections $.\pi$ only match themselves, and so do constructors $\mathsf{c}\,\bar{p}$, but they require successful matching $[\bar{u}\,/\,\bar{p}] \Rightarrow \sigma$ of the arguments. For forced constructors $\lfloor c_1 \rfloor\,\bar{p}$, 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 $\bot$. 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, requiring that $[\bar{e}\,/\,\bar{q}] \Rightarrow \sigma$ is very similar to asking that $\lceil \bar{q} \rceil \sigma = \bar{e}$, but there are two key differences:

1. The matching judgement $[\bar{e}\,/\,\bar{q}] \Rightarrow \sigma$ ignores the forced patterns $\lfloor u \rfloor$ and the constructor names in forced constructor patterns $\lfloor \mathsf{c} \rfloor\,\bar{p}$. This is important to give an efficient implementation of matching as it means we do not have to check equality of arbitrary terms.
2. The matching judgement makes a difference between a mismatch and a *stuck* match. For example, we have $[\mathsf{suc}\,n\,/\,\mathsf{zero}] \Rightarrow \bot$ but $[m + n\,/\,\mathsf{zero}] \not\Rightarrow \bot$. Mere (in)equality cannot distinguish between the two situations.

For the purpose of the evaluation judgement, we would not need to track definite mismatch 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 $[e \,/\, q] \Rightarrow \bot$ or $[\bar{e} \,/\, \bar{q}] \Rightarrow \bot$ alone is not sufficient to derive $[e\,\bar{e} \,/\, q\,\bar{q}] \Rightarrow \bot$, i.e. mismatch does not dominate stuckness, nor does it short-cut matching. Suppose a function and defined by the clauses $\mathsf{true}\ \mathsf{true} \hookrightarrow \mathsf{true}$ and $x\ y \hookrightarrow \mathsf{false}$. If mismatch dominated stuckness, then both open terms $\mathsf{and}\ \mathsf{false}\ y$ and $\mathsf{and}\ x\ \mathsf{false}$ would reduce to 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 Section 2.5 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 (see Theorem 22).[11]

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 η-equality: A lambda abstraction $\lambda x.\, t$ in context Γ can be lifted to the top level and encoded as auxiliary function $\mathsf{f}\,\hat{\Gamma}\,x \hookrightarrow t$. We obtain extensionality (η) by adding the following rule to definitional equality:

$$\frac{\Gamma \vdash t_1 : (x : A) \to B \qquad \Gamma \vdash t_2 : (x : A) \to B \qquad \Gamma(x : A) \vdash t_1\, x = t_2\, x : B}{\Gamma \vdash t_1 = t_2 : (x : A) \to B}\ x \notin \mathrm{dom}(\Gamma)$$

Record expressions: Likewise, a record value $\mathsf{record}\{\bar{\pi} = \bar{v}\}$ in Γ can be turned into an auxiliary definition by copattern matching with clauses $(\mathsf{f}\,\hat{\Gamma}\ .\pi_i \hookrightarrow v_i)_i$. We could add an η-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 η.

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

$$\begin{aligned}&\mathsf{data}\ \mathsf{Vec}\ (A : \mathsf{Set}_\ell)(n : \mathbb{N}) : \mathsf{Set}_\ell\ \mathsf{where}\\ &\quad \mathsf{nil}\ \ : n \equiv_{\mathbb{N}} \mathsf{zero} \to \mathsf{Vec}\ A\ n\\ &\quad \mathsf{cons} : (m : \mathbb{N})(x : A)(xs : \mathsf{Vec}\ A\ m) \to n \equiv_{\mathbb{N}} \mathsf{suc}\ m \to \mathsf{Vec}\ A\ n\end{aligned}$$

Indexed record types can be defined analogously to indexed data types. For example, we can also define $\mathsf{Vec}\ A\ n$ as a record type:

$$\begin{aligned}&\mathsf{record}\ \mathsf{Vec}\ (A : \mathsf{Set}_\ell)(n : \mathbb{N}) : \mathsf{Set}_\ell\ \mathsf{where}\\ &\quad \mathsf{head} : (m : \mathbb{N}) \to n \equiv_{\mathbb{N}} \mathsf{suc}\ m \to A\\ &\quad \mathsf{tail}\ \ : (m : \mathbb{N}) \to n \equiv_{\mathbb{N}} \mathsf{suc}\ m \to \mathsf{Vec}\ A\ m\end{aligned}$$

[11] In a sense, this is opposite to *lazy pattern matching* (Maranget, 1992), which aims to find the right clause with the least amount of matching.

The 'constructors' nil and cons are then defined by:

$$\begin{aligned}
&\mathsf{nil} : \mathsf{Vec}\ A\ \mathsf{zero}\\
&\mathsf{nil}\ .\mathsf{head}\ m\ \emptyset = \mathsf{impossible}\\
&\mathsf{nil}\ .\mathsf{tail}\ m\ \emptyset \ \ = \mathsf{impossible}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{cons} : (n : \mathbb{N})(x : A)(xs : \mathsf{Vec}\ A\ n) \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n)\\
&\mathsf{cons}\ n\ x\ xs\ .\mathsf{head}\ \lfloor n \rfloor\ \mathsf{refl} = x\\
&\mathsf{cons}\ n\ x\ xs\ .\mathsf{tail}\ \lfloor n \rfloor\ \mathsf{refl} \ \ = xs
\end{aligned}$$

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 η. 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.

Other type-level features such as cumulativity or a (predicative or impredicative) Prop universe are orthogonal to the work in this paper and could be added without much trouble. 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 progress:

$$\begin{array}{lll}
Q ::= u & & \text{branch body (splitting done)}\\
\quad \mid \lambda x.\, Q & & \text{bring argument } x \text{ in scope}\\
\quad \mid \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\} & & \text{splitting on projections}\\
\quad \mid \mathsf{case}_x\{c_1\ \hat{\Delta}_1 \mapsto Q_1; \ldots; c_n\ \hat{\Delta}_n \mapsto Q_n\} & & \text{match on data } x\\
\quad \mid \mathsf{case}_x\{\mathsf{refl} \mapsto^{\tau} Q\} & & \text{match on equality proof } x
\end{array} \tag{32}$$

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.

$\boxed{\Sigma;\Gamma \vdash \mathsf{f}\ \bar{q} := Q : C \rightsquigarrow \Sigma'}$ Presupposes: $\Sigma;\Gamma \vdash \mathsf{f}\ \lceil \bar{q} \rceil : C$ and $\mathrm{dom}(\Gamma) = \mathrm{PV}(\bar{q})$.
Checks case tree Q and outputs an extension Σ' of Σ by the clauses represented by "$\mathsf{f}\ \bar{q} \hookrightarrow Q$".

$$\frac{\Sigma;\Gamma \vdash v : C}{\Sigma;\Gamma \vdash \mathsf{f}\ \bar{q} := v : C \rightsquigarrow \Sigma, (\mathsf{clause}\ \Gamma \vdash \mathsf{f}\ \bar{q} \hookrightarrow v : C)}\ \textsc{CtDone}$$

$$\frac{\Sigma;\Gamma \vdash C = (x : A) \to B : \mathsf{Set}_\ell \qquad \Sigma;\Gamma(x : A) \vdash \mathsf{f}\ \bar{q}\ x := Q : B \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash \mathsf{f}\ \bar{q} := \lambda x.\ Q : C \rightsquigarrow \Sigma'}\ \textsc{CtIntro}$$

$$\frac{\begin{array}{c}\Sigma_0;\Gamma \vdash C = \mathsf{R}\ \bar{v} : \mathsf{Set}_\ell \qquad \mathsf{record}\ \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \overline{\pi_i : A_i} \in \Sigma_0 \\ \sigma = [\bar{v}/\Delta, \mathsf{f}\ \lceil \bar{q} \rceil / \mathit{self}] \qquad (\Sigma_{i-1};\Gamma \vdash \mathsf{f}\ \bar{q}\ .\pi_i := Q_i : A_i\sigma \rightsquigarrow \Sigma_i)_{i=1\ldots n}\end{array}}{\Sigma_0;\Gamma \vdash \mathsf{f}\ \bar{q} := \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\} : C \rightsquigarrow \Sigma_n}\ \textsc{CtCosplit}$$

$$\frac{\begin{array}{c}\Sigma_0;\Gamma_1 \vdash A = \mathsf{D}\ \bar{v} : \mathsf{Set}_\ell \qquad \Gamma = \Gamma_1(x : A)\Gamma_2 \qquad \mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \overline{\mathsf{c_i}\ \Delta_i} \in \Sigma_0 \\ \left(\begin{array}{c}\Delta_i' = \Delta_i[\bar{v}/\Delta] \qquad \rho_i = \mathbb{1}_{\Gamma_1} \uplus [\mathsf{c_i}\ \hat{\Delta}_i'/x] \qquad \rho_i' = \rho_i \uplus \mathbb{1}_{\Gamma_2} \\ \Sigma_{i-1};\Gamma_1\Delta_i'(\Gamma_2\rho_i) \vdash \mathsf{f}\ \bar{q}\rho_i' := Q_i : C\rho_i' \rightsquigarrow \Sigma_i\end{array}\right)_{i=1\ldots n}\end{array}}{\Sigma_0;\Gamma \vdash \mathsf{f}\ \bar{q} := \mathsf{case}_x\{\mathsf{c_1}\ \hat{\Delta}_1' \mapsto Q_1; \ldots; \mathsf{c_n}\ \hat{\Delta}_n' \mapsto Q_n\} : C \rightsquigarrow \Sigma_n}\ \textsc{CtSplitCon}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma_1 \vdash A = (u \equiv_B v) : \mathsf{Set}_\ell \qquad \Sigma;\Gamma_1 \vdash_x u =^? v : B \Rightarrow \textsc{yes}(\Gamma_1', \rho, \tau) \\ \rho' = \rho \uplus \mathbb{1}_{\Gamma_2} \qquad \tau' = \tau \uplus \mathbb{1}_{\Gamma_2} \qquad \Sigma;\Gamma_1'(\Gamma_2\rho) \vdash \mathsf{f}\ \bar{q}\rho' := Q : C\rho' \rightsquigarrow \Sigma'\end{array}}{\Sigma;\Gamma_1(x : A)\Gamma_2 \vdash \mathsf{f}\ \bar{q} := \mathsf{case}_x\{\mathsf{refl} \mapsto^{\tau'} Q\} : C \rightsquigarrow \Sigma'}\ \textsc{CtSplitEq}$$

$$\frac{\Sigma;\Gamma_1 \vdash A = (u \equiv_B v) : \mathsf{Set}_\ell \qquad \Sigma;\Gamma_1 \vdash_x u =^? v : B \Rightarrow \textsc{no}}{\Sigma;\Gamma_1(x : A)\Gamma_2 \vdash \mathsf{f}\ \bar{q} := \mathsf{case}_x\{\} : C \rightsquigarrow \Sigma}\ \textsc{CtSplitAbsurdEq}$$

Fig. 7. Declarative (non-algorithmic) typing rules for case trees.

Remark 7 (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 Figure 7.)

4.1 Case tree typing

A case tree Q for a defined constant $\mathsf{f} : A$ is well typed in signature Σ if $\Sigma \vdash \mathsf{f} := Q : A \rightsquigarrow \Sigma'$. In this judgement, Σ is the signature in which case tree Q for function $\mathsf{f} : A$ is well typed, and Σ' is the *output signature* which is Σ extended with the function clauses corresponding to case tree Q. Note that the absence of a local context Γ in this proposition implies that we only use case trees for top-level definitions.[12]

Case tree typing is established by the generalized judgement $\boxed{\Sigma;\Gamma \vdash \mathsf{f}\ \bar{q} := Q : A \rightsquigarrow \Sigma'}$ (Figure 7) that considers a case tree Q for the instance $\mathsf{f}\ \bar{q}$ of the function in a context Γ of the pattern variables of $\bar{q}$. The typing rules presented here capture the well-formedness of the *output* of the elaboration algorithm. In Figure 10, this judgement will be extended to an algorithmic version that takes the user clauses as additional input. We have the following rules for $\Sigma;\Gamma \vdash \mathsf{f}\ \bar{q} := Q : A \rightsquigarrow \Sigma'$:

[12] 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.

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 $\mathsf{f}\,\bar{q}$ and may only refer to the pattern variables Γ of $\bar{q}$. If this is the case, the clause $\mathsf{f}\,\bar{q} \hookrightarrow v$ is added to the signature.

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

CtCosplit If the left-hand side is of record type $\mathsf{R}\,\bar{v}$ with projections π_i, we can do *result splitting* and extend it by copattern $.\pi_i$ for all i. We have $\mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\}$ (where $n \geq 0$) as the corresponding case tree, and we check each subtree Q_i for left-hand side $\mathsf{f}\,\bar{q}\,.\pi_i$ in the signature Σ_{i-1} which includes the clauses for the branches $j < i$. Note that these previous clauses may be needed to check the current case, since we have dependent records (Section 2.2).

CtSplitCon If left-hand side $\mathsf{f}\,\bar{q}$ contains a variable x of data type $D\,\bar{v}$, we can split on x and consider all possible constructors c_i fully applied to fresh variables, generating the case tree $\mathsf{case}_x\{c_1\ \hat{\Delta}'_1 \mapsto Q_1; \ldots; c_n\ \hat{\Delta}'_n \mapsto Q_n\}$. The branch Q_i is checked for a refined left-hand side where x has been substituted by $c_i \hat{\Delta}'_i$ in a context where x has been replaced by the new pattern variables Δ'_i. Note also the threading of signatures as in rule CtCosplit. [13]

The rules CtSplitEq and CtSplitAbsurdEq 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. Unification has three possible outcomes: either it ends in a *positive success* and finds a most general unifier (m.g.u.); then we can build a case tree $\mathsf{case}_x\{\mathsf{refl} \mapsto \cdot\}$ (rule CtSplitEq). Or it ends in a *negative success* with a disunifier; then we may build the case tree $\mathsf{case}_x\{\}$ (rule CtSplitAbsurdEq). Finally, it may end in a *failure* with neither a m.g.u. nor a disunifier, e.g. for equality $y + z \equiv_{\mathbb{N}} y' + z'$; then elaboration fails.

In fact, in our setting, we need a refinement of m.g.u.s we call *strong unifiers*. Compared to the usual notion of m.g.u., a strong unifier has additional restrictions on the computational behaviour of the substitutions between the original context and the reduced ones. 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 8 (Strong unifier)**.** *Let Γ be a well-formed context and u and v be terms such that $\Gamma \vdash u, v : A$. A* strong unifier (Γ', σ, τ) *of u and v consists of a context Γ' and substitutions $\Gamma' \vdash \sigma : \Gamma(x : u \equiv_A v)$ and $\Gamma(x : u \equiv_A v) \vdash \tau : \Gamma'$ such that:*

1. $\Gamma' \vdash x\sigma = \mathit{refl} : u\sigma \equiv_{A\sigma} v\sigma$ *(this implies the definitional equality $\Gamma' \vdash u\sigma = v\sigma : A\sigma$)*
2. $\Gamma' \vdash \tau; \sigma = \mathbb{1}_{\Gamma'} : \Gamma'$ *(recall that $\mathbb{1}_{\Gamma'}$ is the identity substitution $[\hat{\Gamma}' \,/\, \Gamma']$)*
3. *For any context Γ_0 and substitution σ_0 such that $\Gamma_0 \vdash \sigma_0 : \Gamma(x : u \equiv_A v)$ and $\Gamma_0 \vdash x\sigma_0 = \mathit{refl} : u\sigma_0 \equiv_{A\sigma_0} v\sigma_0$, we have $\Gamma_0 \vdash \sigma; \tau; \sigma_0 = \sigma_0 : \Gamma(x : u \equiv_A v)$.*

[13] This threading is not necessary for the current work. However, it allows our approach to be extended to situations where constructors can mention earlier constructors in their types, as for example in the case of path constructors of higher inductive types in homotopy type theory.

$\boxed{\Sigma \vdash Q\sigma\ \bar{e} \longrightarrow v}$ (Σ dropped from rules, used for evaluation $\Sigma \vdash t \longrightarrow^* w$.)

$$\frac{}{v\sigma\ \bar{e} \longrightarrow v\sigma\ \bar{e}} \qquad \frac{Q(\sigma \uplus [u/x])\ \bar{e} \longrightarrow v}{(\lambda x.\ Q)\sigma\ u\ \bar{e} \longrightarrow v} \qquad \frac{Q_i\sigma\ \bar{e} \longrightarrow v}{(\mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\})\sigma\ .\pi_i\ \bar{e} \longrightarrow v}$$

$$\frac{x\sigma \longrightarrow^* \mathsf{c}_i\ \bar{u} \qquad Q_i(\sigma \backslash x \uplus [\bar{u}/\hat{\Delta}_i])\ \bar{e} \longrightarrow v}{(\mathsf{case}_x\{\mathsf{c}_1\ \hat{\Delta}_1 \mapsto Q_1; \ldots; \mathsf{c}_n\ \hat{\Delta}_n \mapsto Q_n\})\sigma\ \bar{e} \longrightarrow v} \qquad \frac{x\sigma \longrightarrow^* \mathsf{refl} \qquad Q(\tau;\sigma)\ \bar{e} \longrightarrow v}{(\mathsf{case}_x\{\mathsf{refl} \mapsto^\tau Q\})\sigma\ \bar{e} \longrightarrow v}$$

Fig. 8. Evaluation of case trees.

Definition 9 (Disunifier)**.** *Let* Γ *be a well-formed context and* $\Gamma \vdash u, v : A$. *A* disunifier *of* u *and* v *is a function* $\Gamma \vdash f : (u \equiv_A v) \to \bot$ *where* $\bot$ *is the empty type.*

Since we use the substitution σ for the construction of the left-hand side of clauses, we require unification to output not just a substitution but a *pattern substitution* ρ. The only properly matching pattern in ρ is $x\rho = \mathsf{refl}$; all the other patterns $y\rho$ are either a forced pattern $\lfloor t \rfloor$ (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:

- A positive success $\boxed{\Sigma; \Gamma \vdash_x u =^? v : A \Rightarrow \textsc{yes}(\Gamma', \rho, \tau)}$ ensures that $x\rho = \mathsf{refl}$ and the triple $(\Gamma', \lceil \rho \rceil, \tau)$ is a strong unifier. Additionally, $\Gamma' \subseteq \Gamma$, $y\tau = y$, and $y\rho = y$ for all $y \in \Gamma'$, and $y\rho$ is a forced pattern for all variables $y \in \Gamma \backslash \Gamma'$.
- A negative success $\boxed{\Sigma; \Gamma \vdash_x u =^? v : A \Rightarrow \textsc{no}}$ ensures that there exists a disunifier of u and v.

Remark 10. During the unification of u with v, each step either instantiates one variable from Γ (e.g. the solution step) or leaves it untouched (e.g. the injectivity step). We thus have the invariant that the variables in Γ' form a subset of the variables in Γ. In effect, the substitution τ makes the variables instantiated by unification go 'out of scope' after a match on refl. This property ceases to hold in a language with η-equality for record types and unification rules for η-expanding a variable such as the ones given by Cockx *et al.* (2016). In particular, τ 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 $\bar{e}$ via the judgement $\boxed{\Sigma \vdash Q\sigma\ \bar{e} \longrightarrow v}$ (Figure 8) with $\sigma = [\,]$. The substitution σ acts as an accumulator, collecting the values for each of the variables introduced by a λ or by the constructor arguments in a $\mathsf{case}_x\{...\}$. In particular, when evaluating a case tree of the form $\mathsf{case}_x\{\mathsf{refl} \mapsto^\tau Q\}$, the substitution τ is used to remove any bindings in σ that correspond to forced patterns.

The operational sematics of case trees give us an efficient way to evaluate functions by pattern matching. Since case trees are guaranteed to be covering, their operational sematics is also essential in the proof of progress (in particular Lemma 19).

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 progress. The goal of this section is to state and prove these properties. First, we need some basic lemmata.

Lemma 11 (Well-typed case trees preserve signature well-formedness)**.** *Let* $\vdash \Sigma$ *be a well-formed signature where definition* $f : A$ *where* $\overline{cls}^{\oplus}$ *is the last declaration in* Σ *and let* Q *be a case tree such that* $\Sigma; \Gamma \vdash f\,\bar{q} := Q : C \rightsquigarrow \Sigma'$ *where* $\Sigma \vdash \Gamma$ *and* $\Sigma; \Gamma \mid f : A \vdash \lceil \bar{q} \rceil : C$*. Then* Σ' *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. In the statement of the lemma, we use the notation $\boxed{[\bar{e} \,/\, \bar{q}] \longrightarrow^* \sigma_\perp}$ to state that $\bar{e} \longrightarrow^* \bar{e}'$ and $[\bar{e}' \,/\, \bar{q}] \Rightarrow \sigma_\perp$ for some $\bar{e}'$.

Lemma 12 (Simulation lemma)**.** *Consider a case tree* Q *such that* $\Sigma_0; \Gamma \vdash f\,\bar{q} := Q : C \rightsquigarrow \Sigma$*, let* σ *be a substitution whose domain is 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* ρ *and copatterns* $\bar{q}'$ *such that* $\text{clause}\ \Delta \vdash f\,\bar{q}\rho\,\bar{q}' \hookrightarrow v : A$ *is in* Σ *but not in* Σ_0 *and* $t = v\theta\,\bar{e}_2$ *where* $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho\,\bar{q}'] \longrightarrow^* \theta$ *and* $\bar{e} = \bar{e}_1\,\bar{e}_2$*.*

Conversely, the signature Σ *is equal to the signature* Σ_0 *extended with clauses of the form* $\text{clause}\ \Delta \vdash f\,\bar{q}\rho\,\bar{q}' \hookrightarrow v : A$*, and for any* σ *and* $\bar{e}_1$ *and* $\bar{e}_2$ *such that* $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho\,\bar{q}'] \Rightarrow \theta$ *we have* $\Sigma \vdash Q\sigma\,\bar{e}_1\,\bar{e}_2 \longrightarrow v\theta\,\bar{e}_2$*.*

Proof We start by proving the first statement by induction on Q:

- In case $Q = v$ we have $\Sigma_0 \vdash Q\sigma\,\bar{e} \longrightarrow v\sigma\,\bar{e}$, and $\Sigma = \Sigma_0$, $\text{clause}\ \Gamma \vdash f\,\bar{q} \hookrightarrow v : A$. Thus, we take $\rho = \mathbb{1}_\Gamma$, $\bar{q}' = \varepsilon$, $v = v$, $\bar{e}_1 = \varepsilon$, and $\bar{e}_2 = \bar{e}$. We clearly have $[\bar{q}\sigma \,/\, \bar{q}] \Rightarrow \sigma$, hence $t = v\sigma\,\bar{e}$.
- In case $Q = \lambda x.\, Q'$, we have $\bar{e} = u\,\bar{e}'$ and $\Sigma \vdash Q(\sigma \uplus [u \,/\, x])\,\bar{e}' \longrightarrow t$. From the induction hypothesis, we know that $\text{clause}\ \Delta \vdash f\,(\bar{q}\,x)\rho\,\bar{q}' \hookrightarrow v : A \in \Sigma$ and $t = v\theta\,\bar{e}_2$ where $[\bar{q}\sigma\ u\,\bar{e}_1 \,/\, (\bar{q}\,x)\rho\,\bar{q}'] \longrightarrow^* \theta$ and $\bar{e}' = \bar{e}_1\,\bar{e}_2$. We can decompose ρ as $\rho' \uplus [p \,/\, x]$, which means that we have $\text{clause}\ \Delta \vdash f\,\bar{q}\rho'\,p\,\bar{q}' \hookrightarrow v : A \in \Sigma$ and $[\bar{q}\sigma'\ u\,\bar{e}_1 \,/\, \bar{q}\rho'\,p\,\bar{q}'] \Rightarrow \theta$, so it suffices to take ρ' as the new ρ and $p\,\bar{q}'$ as the new $\bar{q}'$.
- In case $Q = \text{case}_x\{c_1\ \hat{\Delta}_i' \mapsto Q_1; \ldots; c_n\ \hat{\Delta}_n' \mapsto Q_n\}$, we know that $x\sigma \longrightarrow^* c_i\,\bar{u}$ and $\Sigma \vdash Q_i(\sigma \setminus x \uplus [\bar{u} \,/\, \Delta_i\sigma])\,\bar{e} \longrightarrow t$. Let $\rho_i = \mathbb{1}_{\Gamma_1} \uplus [c_i\ \hat{\Delta}_i' \,/\, x] \uplus \mathbb{1}_{\Gamma_2}$, then by the induction hypothesis we have ρ and $\bar{q}'$ such that $\text{clause}\ \Delta \vdash f\,\bar{q}\rho_i\rho\,\bar{q}' \hookrightarrow v : A \in \Sigma$ and $t = v\theta\,\bar{e}_2$ where $[\bar{q}\rho_i(\sigma \setminus x \uplus [\bar{u} \,/\, \Delta_i\sigma])\,\bar{e}_1 \,/\, \bar{q}\rho_i\rho\,\bar{q}'] \longrightarrow^* \theta$ and $\bar{e} = \bar{e}_1\,\bar{e}_2$. From the definition of matching, it follows that also $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho_i\rho\,\bar{q}'] \Rightarrow \theta$. Thus, we finish this case by taking $\rho_i; \rho$ as the new ρ (and keep $\bar{q}'$ the same).

- In case $Q = \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\}$, we have $\bar{e} = .\pi_i\, \bar{e}'$ and $\Sigma \vdash Q_i\sigma\ \bar{e}' \longrightarrow t$. From the induction hypothesis, we know that $\mathsf{clause}\ \Delta \vdash \hookrightarrow \mathsf{f}\,\bar{q}\rho\ .\pi_i\,\bar{q}' : vA \in \Sigma$ and $t = v\theta\ \bar{e}_2$ where $[\bar{q}\sigma\ .\pi_i\,\bar{e}_1 \,/\, \bar{q}\rho\ .\pi_i\,\bar{q}'] \longrightarrow^* \theta$ and $\bar{e}' = \bar{e}_1\ \bar{e}_2$. Hence, it suffices to take $.\pi_i\,\bar{q}'$ as the new $\bar{q}'$ (and keep ρ the same).
- In case $Q = \mathsf{case}_x\{\mathsf{refl} \mapsto^{\tau'} Q'\}$, we have $x\sigma \longrightarrow^* \mathsf{refl}$ and $\Sigma \vdash Q'(\tau'; \sigma)\,\bar{e} \longrightarrow t$. From the induction hypothesis, we know that $\mathsf{clause}\ \Delta \vdash \mathsf{f}\,\bar{q}\rho'\rho \hookrightarrow v : A \in \Sigma$ and and $t = v\theta\ \bar{e}_2$ where $[\bar{q}\rho'\tau'\sigma\ \bar{e}_1 \,/\, \bar{q}\rho'\rho\ \bar{q}'] \longrightarrow^* \theta$ and $\bar{e} = \bar{e}_1\ \bar{e}_2$. Since ρ' and τ' are produced by unification, we have that $x\rho' = \mathsf{refl}$ and for each pattern variable y of $\bar{q}$ other than x, either $y\rho' = \lfloor s \rfloor$ or $y\rho' = y$ and $y\tau' = y$. It then follows from the definition of matching that $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho'\rho\ \bar{q}'] \Rightarrow \theta$. Hence, we take $\rho'; \rho$ as the new ρ (and keep $\bar{q}'$ the same).
- There are no evaluation rules for $Q = \mathsf{case}_x\{\}$ so this case is impossible.

In the other direction, we start again by induction on Q:

- In case $Q = v$, we have the single clause $\mathsf{clause}\ \Gamma \vdash \mathsf{f}\,\bar{q} \hookrightarrow v : A$ which is of the right form with $\rho = \mathbb{1}_\Gamma$ and $\bar{q}' = \varepsilon$. If $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}] \Rightarrow \theta$, then we have $\sigma = \theta$ and $\bar{e}_1 = \varepsilon$, so $\Sigma \vdash Q\sigma\ \bar{e}_1\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$.
- In case $Q = \lambda x.\, Q'$, we get from the induction hypothesis that any clause in $\Sigma \backslash \Sigma_0$ is of the form $\mathsf{clause}\ \Delta \vdash \mathsf{f}\,(\bar{q}\,x)\rho\,\bar{q}' \hookrightarrow v : A$, which is of the right form if we take $\rho' = \rho \backslash x$ as the new ρ and $\bar{q}'' = x\rho\,\bar{q}'$ as the new $\bar{q}'$. Moreover, if $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho'\,\bar{q}''] \Rightarrow \theta$, then $\bar{e}_1 = u\,\bar{e}_1'$ and $[(\bar{q}\,x)(\sigma \uplus [u \,/\, x])\,\bar{e}_1' \,/\, (\bar{q}\,x)\rho\,\bar{q}'] \Rightarrow \theta$. The induction hypothesis gives us that $\Sigma \vdash Q'(\sigma \uplus [u \,/\, x])\,\bar{e}_1'\,\bar{e} \longrightarrow v\theta\ \bar{e}_2$, hence also $\Sigma \vdash Q\sigma\ \bar{e}_1\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$.
- In case $Q = \mathsf{case}_x\{c_1\ \hat{\Delta}_i' \mapsto Q_1; \ldots; c_n\ \hat{\Delta}_n' \mapsto Q_n\}$, we get from the induction hypothesis that any clause in $\Sigma \backslash \Sigma_0$ is of the form $\mathsf{clause}\ \Delta \vdash \mathsf{f}\,\bar{q}\rho_i\rho\,\bar{q}' \hookrightarrow v : A$ for some $\rho_i = \mathbb{1}_{\Gamma_1} \uplus [c_i\ \hat{\Delta}_i' \,/\, x] \uplus \mathbb{1}_{\Gamma_2}$. This is of the right form if we take $\rho' = \rho_i\rho$ as the new ρ (and keep $\bar{q}'$ the same). Moreover, if $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho_i\rho\ \bar{q}'] \Rightarrow \theta$, then we have $x\sigma = \mathsf{c}_i\ \bar{u}$ from the definition of matching. Let $\sigma' = \sigma \backslash x \uplus [\bar{u} \,/\, \Delta_i\sigma]$, then we also have $[\bar{q}\rho_i\sigma'\ \bar{e}_1 \,/\, \bar{q}\rho_i\rho\ \bar{q}'] \Rightarrow \theta$. From the induction hypothesis, it now follows that $\Sigma \vdash Q_i\sigma'\ \bar{e}_1\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$, hence also $\Sigma \vdash Q\sigma\ \bar{e}_1\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$.
- In case $Q = \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\}$, we get from the induction hypothesis that any clause in $\Sigma \backslash \Sigma_0$ is of the form $\mathsf{clause}\ \Delta \vdash \mathsf{f}\,\bar{q}\rho\ .\pi_i\,\bar{q}' \hookrightarrow v : A$. This is of the right form if we take $\bar{q}'' = .\pi_i\,\bar{q}'$ as the new $\bar{q}'$ (and keep ρ the same). Moreover, if $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho\ .\pi_i\,\bar{q}'] \Rightarrow \theta$, then $\bar{e}_1 = .\pi_i\,\bar{e}_1'$. The induction hypothesis gives us that $\Sigma \vdash Q_i\sigma\ \bar{e}_1'\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$, hence also $\Sigma \vdash Q\sigma\ \bar{e}_1\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$.
- In case $Q = \mathsf{case}_x\{\mathsf{refl} \mapsto^{\tau'} Q\}'$, we get from the induction hypothesis that any clause in $\Sigma \backslash \Sigma_0$ is of the form $\mathsf{clause}\ \Delta \vdash \mathsf{f}\,\bar{q}\rho'\rho \hookrightarrow v : A$ where ρ' and τ' are produced by unification. This is of the right form if we take $\rho'' = \rho'; \rho$ as the new ρ (and keep $\bar{q}'$ the same). Moreover, if $[\bar{q}\sigma\ \bar{e}_1 \,/\, \bar{q}\rho'\rho\ \bar{q}'] \Rightarrow \theta$, then we have $x\sigma = \mathsf{refl}$ from the definition of matching. Let $\sigma' = \tau'; \sigma$, then we have $x\rho'\sigma' = \mathsf{refl}$ and for all other pattern variables y of $\bar{q}$, either $y\rho'$ is a forced pattern or $y\rho' = y$ and $y\sigma' = y\sigma$. By matching, it follows that also $[\bar{q}\rho'\sigma'\ \bar{e}_1 \,/\, \bar{q}\rho'\rho\ \bar{q}'] \Rightarrow \theta$. From the induction hypothesis, it now follows that $\Sigma \vdash Q'\sigma'\ \bar{e}_1\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$, hence also $\Sigma \vdash Q\sigma\ \bar{e}_1\ \bar{e}_2 \longrightarrow v\theta\ \bar{e}_2$.
- In case $Q = \mathsf{case}_x\{\}$, we have $\Sigma = \Sigma_0$ so there are no new clauses to worry about.

□

Before adding a clause $\mathsf{f}\,\bar{q} \hookrightarrow v$ 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 $[\bar{e} \,/\, \bar{q}] \Rightarrow \sigma$ but $\lceil \bar{q} \rceil \sigma \neq \bar{e}$. This is captured in the notion of a *respectful pattern* (Goguen *et al.*, 2006). Intuitively, a pattern is respectful if any well-typed term that matches the accessible part of the pattern also matches the inaccessible parts. Here, we generalize the definition to the case where we do not yet know that all reductions in the signature are necessarily type-preserving. This requires us to first define respectfulness of a *signature*.

Definition 13. *A signature* Σ *is* respectful for $\Sigma \vdash u \longrightarrow v$ *if* $\Sigma; \Gamma \vdash u : A$ *implies* $\Sigma; \Gamma \vdash u = v : A$. *A signature* Σ *is respectful if it is respectful for all derivations of* $\Sigma \vdash u \longrightarrow v$.

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

Definition 14 (Respectful copatterns)**.** *Let* $\bar{q}$ *be a list of copatterns such that* $\Sigma; \Delta \mid u : A \vdash \lceil \bar{q} \rceil : C$ *where u and A are closed (i.e. do not depend on* Δ*). We call* $\bar{q}$ respectful *in signature* Σ *if the following holds: for any signature extension* $\Sigma \subseteq \Sigma'$ *and any eliminations* $\Sigma'; \Gamma \mid u : A \vdash \bar{e} : C$ *such that* $[\bar{e} \,/\, \bar{q}] \Rightarrow \sigma$, *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 Σ and $\Sigma \subseteq \Sigma'$, then $\bar{q}$ is also respectful in Σ'.

Lemma 15 (Signatures with respectful clauses are respectful)**.** *If* Σ *is a well-formed signature such that all clauses in* Σ *have respectful copatterns in* Σ, *then* Σ *is respectful.*

Proof By induction on the derivation of $\Sigma \vdash u \longrightarrow v$. Assume $\mathsf{clause}\,\Delta \vdash \mathsf{f}\,\bar{q} \hookrightarrow v : C \in \Sigma$ and $[\bar{e} \,/\, \bar{q}] \Rightarrow \sigma$ for well-typed eliminations $\Sigma; \Gamma \mid \mathsf{f} : C \vdash \bar{e} : A$, then we have to prove that $\Sigma; \Gamma \vdash \mathsf{f}\,\bar{e} = v\sigma : A$. Since $\bar{q}$ is respectful, we have $\Sigma; \Gamma \mid \mathsf{f} : C \vdash \bar{q}\sigma = \bar{e} : A$. It follows that $\Sigma; \Gamma \vdash \mathsf{f}\,\bar{q}\sigma = \mathsf{f}\,\bar{e} : A$, hence also $\Sigma; \Gamma \vdash \mathsf{f}\,\bar{e} = v\sigma : A$ by the β-rule for definitional equality. □

Lemma 16 (Well-typed case trees have respectful clauses)**.** *Consider a respectful signature* Σ_0 *and a case tree Q such that* $\Sigma_0; \Gamma \vdash \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma$ *and* $\bar{q}$ *is respectful in* Σ_0. *Then all clauses in* $\Sigma \setminus \Sigma_0$ *have respectful patterns in* Σ.

Proof By induction on the derivation of $\Sigma_0; \Gamma \vdash \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma$:

- In case $Q = v$, we have a single new clause $\mathsf{clause}\,\Gamma \vdash \mathsf{f}\,\bar{q} \hookrightarrow v : C$. Since $\bar{q}$ is respectful in Σ_0 by assumption, it is also respectful in $\Sigma = \Sigma_0, \mathsf{clause}\,\Gamma \vdash \mathsf{f}\,\bar{q} \hookrightarrow v : C$.
- In case $Q = \lambda x.\, Q'$, we know from the typing rule of $\lambda x.$ that $\Sigma_0; \Gamma \vdash C = (x : A') \to B' : \mathsf{Set}_\ell$. and $\Sigma_0; \Gamma(x : A) \vdash \mathsf{f}\,\bar{q}\,x := Q' : B \rightsquigarrow \Sigma$. Since $\bar{q}$ is respectful, it follows that $\bar{q}\,x$ is also respectful, so the result follows from the induction hypothesis.

- In case $Q = \mathsf{case}_x\{c_1\ \hat{\Delta}_i' \mapsto Q_1; \ldots; c_n\ \hat{\Delta}_n' \mapsto Q_n\}$, the typing rule for $\mathsf{case}_x\{\}$ tells us that $\Gamma = \Gamma_1(x : A)\Gamma_2$ and $\Sigma_0; \Gamma_1 \vdash A = \mathsf{D}\ \bar{v} : \mathsf{Set}_\ell$. Moreover, it tells us for $n = 1, \ldots, n$ that $\Sigma_{i-1}; \Gamma_1\Delta_i'\Gamma_2\rho_i \vdash \mathsf{f}\ \bar{q}\rho_i := Q_i : C\rho_i \rightsquigarrow \Sigma_i$ where $\mathsf{constructor}\ c_i\ \Delta_i : \mathsf{D}\ \Delta \in \Sigma_0$ and $\Delta_i' = \Delta_i[\bar{v}\ /\ \Delta]$ and $\rho_i = [c_i\ \hat{\Delta}_i'\ /\ x]$. Since $\bar{q}$ is respectful, so is $\bar{q}\rho_i$, so the result follows from the induction hypothesis.
- In case $Q = \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\}$, the typing rule for $\mathsf{record}\{\}$ tells us that $\Sigma_0; \Gamma \vdash C = \mathsf{R}\ \bar{v} : \mathsf{Set}_\ell$. We also get that $\Sigma_{i-1}; \Gamma \vdash \mathsf{f}\ \bar{q}\ .\pi_i := Q_i : A_i[\bar{v}\ /\ \Delta, \mathsf{f}\ \lceil\bar{q}\rceil\ /\ x] \rightsquigarrow \Sigma_i$ where $\mathsf{projection}\ x : \mathsf{R}\ \Delta \vdash\ .\pi_i : A_i \in \Sigma_0$. Since $\bar{q}$ is respectful, so is $\bar{q}\ .\pi_i$, so the result follows from the induction hypothesis.
- In case $Q = \mathsf{case}_x\{\mathsf{refl} \mapsto^\tau Q\}'$, the the typing rule tells us that $\Gamma = \Gamma_1(x : A)\Gamma_2$ and $\Sigma_0; \Gamma_1 \vdash A = s \equiv_E t : \mathsf{Set}_\ell$. We also have that $\Sigma_0; \Gamma_1 \vdash_x s =^? t : E \Rightarrow \textsc{yes}(\Gamma_1', \rho, \tau)$ and $\Sigma_0; \Gamma_1'\Gamma_2\rho \vdash \mathsf{f}\ \bar{q}\rho' := Q' : C\rho' \rightsquigarrow \Sigma$ where $\rho' = \rho \uplus \mathbb{1}_{\Gamma_2}$. Since $\bar{q}$ is respectful and ρ is a strong unifier (Definition 8), $\bar{q}\rho'$ is also respectful, so the result follows from the induction hypothesis.
- The typing rule for $Q = \mathsf{case}_e\{\}$ does not add any new clauses.

□

Theorem 17 (Type preservation)**.** *If all functions in a signature Σ are given by well-typed case trees, then Σ is respectful.*

Proof This is a direct consequence of the previous two lemmata. □

Theorem 18 (Progress)**.** *Let Σ be a well-formed signature where all functions are given by well-typed case trees. Then for any function definition $f : A \in \Sigma$ and closed eliminations $\Sigma \mid f : A \vdash \bar{e} : B$ such that B is not definitionally equal to a function type or a record type, we have $\Sigma \vdash f\bar{e} \longrightarrow v$ for some v.*

In particular, this theorem tells us that evaluation of a function defined by a well-typed case tree applied to closed arguments can never get stuck. To prove this theorem, we first prove the following auxiliary lemma.

Lemma 19. *Let Σ be a well-formed signature where all functions other than f are given by well-typed case trees. Let further Q be a case tree such that $\Sigma_0; \Gamma \vdash f\bar{q} := Q : C \rightsquigarrow \Sigma$, and let $\Sigma \vdash \sigma_0 : \Gamma$ be a (closed) substitution and $\Sigma \mid f\bar{q}\sigma_0 : C\sigma_0 \vdash \bar{e} : B$ be (closed) eliminations such that B is not definitionally equal to a function type or a record type, and σ_0 and $\bar{e}$ do not mention f. Then one of the following holds*

1. *$\Sigma \vdash x\sigma_0 \longrightarrow v$ for some variable x in Γ, or*
2. *$\Sigma \vdash \bar{e} \longrightarrow \bar{e}'$, or*
3. *$\Sigma \vdash Q\sigma_0\ \bar{e} \longrightarrow v$.*

From this lemma, the theorem follows directly by induction on well-formedness of Σ and applying the lemma for each function with $\Gamma = \varepsilon$, $\sigma_0 = [\,]$, and $\bar{q} = \varepsilon$.

Proof By induction on Q:

- If $Q = v$, we have $\mathsf{clause}\ \Gamma \vdash \mathsf{f}\ \bar{q} \hookrightarrow v : C \in \Sigma$ so we are in the third case.
- If $Q = \lambda x.\ Q'$, we have $\Sigma_0; \Gamma \vdash C = (x : A') \to B' : \mathsf{Set}_\ell$ and $\Sigma_0; \Gamma(x : A') \vdash \mathsf{f}\ \bar{q}\ x := Q' : B' \rightsquigarrow \Sigma$ from the typing rule of $\lambda x.$. Hence, we have $\Sigma; \Gamma \vdash C\sigma_0 = (x : A'\sigma_0) \to$

$B'\sigma_0 : \mathsf{Set}_\ell$, so $\bar{e} = w\,\bar{e}'$ for some term $\Sigma \vdash w : A'\sigma_0$ and eliminations $\Sigma \mid \mathsf{f}\,\bar{q}\sigma_0\,w : B'(\sigma_0 \uplus [w \,/\, x]) \vdash \bar{e}' : B$. By induction, we now have that either σ_0 or $\bar{e}$ takes a step, or else $\Sigma \vdash Q'(\sigma_0 \uplus [w \,/\, x])\,\bar{e}' \longrightarrow v$, hence also $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow v$.

- If $Q = \mathsf{case}_x\{c_1\ \hat{\Delta}_i' \mapsto Q_1; \ldots; c_n\ \hat{\Delta}_n' \mapsto Q_n\}$, we have $x : \mathsf{D}\,\bar{v} \in \Gamma$, hence either $x\sigma_0$ takes a step (so we are in the first case) or else $x\sigma_0$ is a weak head normal form $c_i\,\bar{u}$ for some constructor c_i of D. In the latter case, we have by induction that either $\sigma_0 \uplus [\bar{u} \,/\, \Delta_i\sigma]$ or $\bar{e}$ takes a step, or else $\Sigma \vdash Q_i(\sigma_0 \uplus [\bar{u} \,/\, \Delta_i\sigma])\,\bar{e} \longrightarrow v$, hence also $\Sigma \vdash Q\sigma\,\bar{e} \longrightarrow v$.
- If $Q = \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\}$, we have $\Sigma_0; \Gamma \vdash C = \mathsf{R}\,\bar{v} : \mathsf{Set}_\ell$. Hence, we have $\Sigma_0 \vdash C\sigma_0 = \mathsf{R}\,\bar{v}\sigma_0 : \mathsf{Set}_\ell$, so $\bar{e} = .\pi_i\,\bar{e}'$ for some field π_i of R. By induction either σ_0 or $\bar{e}$ takes a step, or else we get a v such that $\Sigma \vdash Q_i\sigma_0\,\bar{e}' \longrightarrow v$, hence also $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow v$.
- If $Q = \mathsf{case}_x\{\mathsf{refl} \mapsto^\tau Q'\}$, we have $x : u \equiv_E v \in \Gamma$, hence either $x\sigma_0$ takes a step (so we are in the first case) or else $x\sigma_0 = \mathsf{refl}$. From the inductive hypothesis together follows that either $\tau; \sigma_0$ or $\bar{e}$ takes a step, or else $\Sigma \vdash Q'(\tau; \sigma_0)\,\bar{e} \longrightarrow v$. By construction, $y\tau = y$ for each variable y in Γ' (see Remark 10), so if $\tau; \sigma_0$ takes a step then so does σ_0. On the other hand, if $\Sigma \vdash Q'(\tau; \sigma_0)\,\bar{e} \longrightarrow v$, then we also have $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow v$ so we are in the third case.
- If $Q = \mathsf{case}_x\{\}$, we have $x : u \equiv_E v \in \Gamma$, so either $x\sigma_0$ takes a step or it is equal to refl. But $u \equiv_E v$ is equivalent to the empty type by unification, so the latter case is impossible.

□

5 Elaboration

In the previous two sections, we have described a core language with inductive data types, 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 data types, record types, and functions by dependent (co)pattern matching. In this section, we show how to elaborate a programme in this surface language to a well-formed signature in the core language.

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 Figure 5. Consequentially, if $\vdash \Sigma$ and $\Sigma \vdash \mathit{decl} \rightsquigarrow \Sigma'$, 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:

$\boxed{\Sigma \vdash \mathit{decl} \rightsquigarrow \Sigma'}$ Presupposes: $\vdash \Sigma$. Entails: $\vdash \Sigma'$.

$$\frac{\Sigma \vdash \Delta \qquad (\Sigma, \mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell) \mid \mathsf{D}\ \Delta : \mathsf{Set}_\ell \vdash \overline{con} \rightsquigarrow \Sigma'}{\Sigma \vdash (\mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \overline{con}) \rightsquigarrow \Sigma'}$$

$$\frac{\Sigma \vdash \Delta \qquad (\Sigma, \mathsf{record}\ \mathsf{R}\ \Delta : \mathsf{Set}_\ell) \mid \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_\ell \vdash \overline{fld} \rightsquigarrow \Sigma'}{\Sigma \vdash (\mathsf{record}\ \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \overline{fld}) \rightsquigarrow \Sigma'}$$

$$\frac{\Sigma \vdash A \qquad (\Sigma, \mathsf{definition}\ \mathsf{f} : A) \vdash P \mid \mathsf{f} := Q : A \rightsquigarrow \Sigma'}{\Sigma \vdash (\mathsf{definition}\ \mathsf{f} : A\ \mathsf{where}\ P) \rightsquigarrow \Sigma'}$$

$\boxed{\Sigma \mid \mathsf{D}\ \Delta : \mathsf{Set}_\ell \vdash \overline{con} \rightsquigarrow \Sigma'}$
Presupposes: $\vdash \Sigma$ and $\Sigma \vdash \mathsf{D} : \Delta \to \mathsf{Set}_\ell$. Entails: $\vdash \Sigma'$.

$$\frac{}{\Sigma \mid \mathsf{D}\ \Delta : \mathsf{Set}_\ell \vdash \varepsilon \rightsquigarrow \Sigma}$$

$$\frac{\Sigma; \Delta \vdash_\ell \Delta_\mathsf{c} \qquad (\Sigma, \mathsf{constructor}\ \mathsf{c}\ \Delta_\mathsf{c} : \mathsf{D}\ \Delta) \mid \mathsf{D}\ \Delta : \mathsf{Set}_\ell \vdash \overline{con} \rightsquigarrow \Sigma'}{\Sigma \mid \mathsf{D}\ \Delta : \mathsf{Set}_\ell \vdash \mathsf{c}\ \Delta_\mathsf{c}, \overline{con} \rightsquigarrow \Sigma'}$$

$\boxed{\Sigma \mid \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_\ell \vdash \overline{fld} \rightsquigarrow \Sigma'}$
Presupposes: $\vdash \Sigma$ and $\Sigma \vdash \mathsf{R} : \Delta \to \mathsf{Set}_\ell$. Entails: $\vdash \Sigma'$.

$$\frac{}{\Sigma \mid \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_\ell \vdash \varepsilon \rightsquigarrow \Sigma}$$

$$\frac{\Sigma; \Delta(\mathit{self} : \mathsf{R}\ \hat{\Delta}) \vdash A : \mathsf{Set}_{\ell'} \qquad \ell' \leq \ell \qquad (\Sigma, \mathsf{projection}\ \mathit{self} : \mathsf{R}\ \Delta \vdash .\pi : A) \mid \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_\ell \vdash \overline{fld} \rightsquigarrow \Sigma'}{\Sigma \mid \mathit{self} : \mathsf{R}\ \Delta : \mathsf{Set}_\ell \vdash \pi : A, \overline{fld} \rightsquigarrow \Sigma'}$$

Fig. 9. Declarative (non-algorithmic) rules for checking declarations of data types, record types, and defined symbols.

- A signature Σ containing previous declarations, as well as clauses for the branches of the case tree that have already been checked.
- A context Γ containing the types of the pattern variables: $\mathrm{dom}(\Gamma) = \mathrm{PV}(\bar{q})$.
- 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 Σ' extending Σ with new clauses and a well-typed case tree Q such that $\Sigma; \Gamma \vdash \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma'$.

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 $[E]\bar{q} \hookrightarrow \mathit{rhs}$, where E is an (unordered) set of constraints $\{w_k \stackrel{?}{/} p_k : A_k \mid k = 1 \ldots l\}$ 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 $\bar{q} \hookrightarrow \mathit{rhs}$.

Table 2. *List of judgements and definitions used by the elaboration algorithm.*

Judgement/definition	Explanation	Rules
$\Sigma; \Gamma \vdash P \mid \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma'$	User input P is elaborated to the case tree Q which implements $\mathsf{f}\,\bar{q}$, Σ' extends Σ with the clauses of Q.	Figure 10
$\Sigma; \Gamma \vdash E \Rightarrow \textsc{solved}(\sigma)$	The constraints E are solved by substitution σ.	Figure 11
$P\,(x : A)$	Update user input P by introducing new argument x.	Figure 12
$P\,.\pi$	Update user input P by applying projection $.\pi$.	Figure 13
$\Sigma \vdash P\sigma \Rightarrow P'$	Applying substitution σ to user input P produces P'.	Figure 14
$\Sigma \vdash \bar{v} \mathbin{/^?} \bar{p} : \Delta \Rightarrow E_\bot$	Constraints $\bar{v} \mathbin{/^?} \bar{p}$ can be simplified to $E_\bot$.	Figure 15
$\Sigma; \Gamma \vdash \emptyset : A$	Type A is a caseless type.	Figure 16

Elaboration of an lhs problem to a well-typed case tree is defined by the judgements in Table 2, which are explained further in the following paragraphs. The main elaboration judgement $\boxed{\Sigma; \Gamma \vdash P \mid \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma'}$ is defined in Figure 10. This judgement is designed as an algorithmic version of the typing judgement for case trees $\Sigma; \Gamma \vdash \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma'$, where the extra user input P guides the construction of the case tree. In particular, the clauses in P serve as the user-provided definition of $\mathsf{f}\,\bar{q}$. Each of the rules in Figure 10 is a refined version of one of the rules in Figure 7, so any case tree produced by this elaboration is well typed by construction. In particular, since well-typed case trees are guaranteed to be covering, this judgement doubles as a coverage checking algorithm for the clauses in P.

To check a definition of $\mathsf{f} : A$ with clauses $\bar{q}_i \hookrightarrow rhs_i$ for $i = 1 \ldots n$, the algorithm starts with $\Gamma = \varepsilon$, $\bar{q} = \varepsilon$, and $P = \{\bar{q}_i \hookrightarrow rhs_i \mid i = 1 \ldots n\}$. If we obtain $\Sigma; \Gamma \vdash P \mid \mathsf{f} := Q : A \rightsquigarrow \Sigma'$, 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, $\Sigma \vdash \Gamma$ is a well-formed context, and $\Sigma; \Gamma \vdash \mathsf{f}\,\lceil\bar{q}\rceil : C$. It also maintains the invariant that for each constraint $w_k \mathbin{/^?} p_k : A_k$ in the lhs problem, we have $\Sigma; \Gamma \vdash w_k : A_k$.

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

- After each step, the algorithm uses $\boxed{\Sigma; \Gamma \vdash E \Rightarrow \textsc{solved}(\sigma)}$ (Figure 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 σ assigning a well-typed value to each of the user-written pattern variables.
- After introducing a new variable, the algorithm uses $\boxed{P\,(x : A)}$ (Figure 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 $\boxed{P\,.\pi}$ (Figure 13) to partition the clauses in the lhs problem according to the projection they belong to.
- After a case split on a data type or an equality proof, the algorithm uses $\boxed{\Sigma \vdash P\sigma \Rightarrow P'}$ (Figure 14) to refine the constraints in the problem. It makes use of the judgements $\boxed{\Sigma \vdash v \mathbin{/^?} p : A \Rightarrow E_\bot}$ and $\boxed{\Sigma \vdash \bar{v} \mathbin{/^?} \bar{p} : \Delta \Rightarrow E_\bot}$ (Figure 15) to simplify the constraints if possible, and to filter out the clauses that definitely do not match the current branch (see Section 2.1 for an example).

$\boxed{\Sigma;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := Q : C \rightsquigarrow \Sigma'}$ In all rules $P = \{[E_i]\bar{q}_i \hookrightarrow rhs_i \mid i = 1 \ldots m\}$.

Presupposes: $\Sigma;\Gamma \vdash \mathsf{f}\ \lceil \bar{q} \rceil : C$ and $\mathrm{dom}(\Gamma) = \mathrm{PV}(\bar{q})$.
Entails: $\Sigma;\Gamma \vdash \mathsf{f}\ \bar{q} := Q : C \rightsquigarrow \Sigma'$.

$$\frac{\bar{q}_1 = \varepsilon \qquad \Sigma;\Gamma \vdash E_1 \Rightarrow \textsc{solved}(\sigma) \qquad rhs_1 = v \qquad \Sigma;\Gamma \vdash v\sigma : C}{\Sigma;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := v\sigma : C \rightsquigarrow \Sigma, \mathsf{clause}\ \Gamma \vdash \mathsf{f}\ \bar{q} \hookrightarrow v\sigma : C}\ \textsc{Done}$$

$$\frac{\bar{q}_1 = p\ \bar{q}'_1 \qquad \Sigma \vdash C \longrightarrow^* (x : A) \to B \qquad \Sigma;\Gamma(x : A) \vdash P\ (x : A) \mid \mathsf{f}\ \bar{q}\ x := Q : B \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := \lambda x.\ Q : C \rightsquigarrow \Sigma'}\ \textsc{Intro}$$

$$\frac{\bar{q}_1 = .\pi_i\ \bar{q}'_1 \qquad \Sigma \vdash C \longrightarrow^* \mathsf{R}\ \bar{v} \qquad \mathsf{record}\ self : \mathsf{R}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \overline{\pi_i : A_i} \in \Sigma_0 \qquad (\Sigma_{i-1};\Gamma \vdash P\ .\pi_i \mid \mathsf{f}\ \bar{q}\ .\pi_i := Q_i : A_i[\bar{v}/\Delta, \mathsf{f}\ \lceil \bar{q} \rceil / self] \rightsquigarrow \Sigma_i)_{i=1\ldots n}}{\Sigma;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\} : C \rightsquigarrow \Sigma_n}\ \textsc{CoSplit}$$

$$\frac{\bar{q}_1 = \emptyset \qquad m = 1 \qquad \Sigma \vdash C \longrightarrow^* \mathsf{R}\ \bar{v} \qquad \mathsf{record}\ \_ : \mathsf{R}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \varepsilon \in \Sigma \qquad rhs_1 = \mathsf{impossible}}{\Sigma;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := \mathsf{record}\{\} : C \rightsquigarrow \Sigma}\ \textsc{CoSplitEmpty}$$

$$\frac{(x\ /^?\ \mathsf{c_j}\ \bar{p} : A) \in E_1 \qquad \Sigma \vdash A \longrightarrow^* \mathsf{D}\ \bar{v} \qquad \Gamma = \Gamma_1(x : A)\Gamma_2 \qquad \mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \overline{\mathsf{c_i}\ \Delta_i} \in \Sigma_0 \qquad \left(\begin{array}{c} \Delta'_i = \Delta_i[\bar{v}/\Delta] \qquad \rho_i = \mathbb{1}_{\Gamma_1} \uplus [\mathsf{c_i}\ \hat{\Delta}'_i/x] \\ \rho'_i = \rho_i \uplus \mathbb{1}_{\Gamma_2} \qquad \Sigma_n \vdash P\rho'_i \Rightarrow P_i \\ \Sigma_{i-1};\Gamma_1\Delta'_i(\Gamma_2\rho_i) \vdash P_i \mid \mathsf{f}\ \bar{q}\rho'_i := Q_i : C\rho'_i \rightsquigarrow \Sigma_i \end{array}\right)_{i=1\ldots n}}{\Sigma_0;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := \mathsf{case}_x\{\mathsf{c_1}\ \hat{\Delta}'_1 \mapsto Q_1; \ldots; \mathsf{c_n}\ \hat{\Delta}'_n \mapsto Q_n\} : C \rightsquigarrow \Sigma_n}\ \textsc{SplitCon}$$

$$\frac{(x\ /^?\ \mathsf{refl} : A) \in E_1 \qquad \Sigma \vdash A \longrightarrow^* u \equiv_B v \qquad \Gamma = \Gamma_1(x : A)\Gamma_2 \qquad \Sigma;\Gamma_1 \vdash_x u =^? v : B \Rightarrow \textsc{yes}(\Gamma'_1, \rho, \tau) \qquad \rho' = \rho \uplus \mathbb{1}_{\Gamma_2} \qquad \tau' = \tau \uplus \mathbb{1}_{\Gamma_2} \qquad \Sigma \vdash P\rho' \Rightarrow P' \qquad \Sigma;\Gamma'_1(\Gamma_2\rho) \vdash P' \mid \mathsf{f}\ \bar{q}\rho' := Q : C\rho' \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := \mathsf{case}_x\{\mathsf{refl} \mapsto^{\tau'} Q\} : C \rightsquigarrow \Sigma'}\ \textsc{SplitEq}$$

$$\frac{(x\ /^?\ \emptyset : A) \in E_1 \qquad \Sigma;\Gamma \vdash \emptyset : A \qquad rhs_1 = \mathsf{impossible}}{\Sigma;\Gamma \vdash P \mid \mathsf{f}\ \bar{q} := \mathsf{case}_x\{\} : C \rightsquigarrow \Sigma}\ \textsc{SplitEmpty}$$

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

- To check an absurd pattern $\emptyset$, the algorithm uses $\boxed{\Sigma;\Gamma \vdash \emptyset : A}$ (Figure 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: data types $\mathsf{D}\ \bar{v}$ with no constructors, and identity types $u \equiv_A v$ where $\Sigma;\Gamma \vdash_x u =^? v : A \Rightarrow \textsc{no}$.

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

Done applies when the first user clause in P has no more copatterns and all its constraints are solved according to $\Sigma;\Gamma \vdash E \Rightarrow \textsc{solved}(\sigma)$. If this is the case, then construction of the case tree is finished, adding the clause $\mathsf{clause}\ \Gamma \vdash \mathsf{f}\ \bar{q} \hookrightarrow v\sigma : C$ to the signature.

$\boxed{\Sigma;\Gamma \vdash E \Rightarrow \text{SOLVED}(\sigma)}$

$$\frac{([w_k \,/\, p_k] \Rightarrow \sigma_k)_{k=1\ldots n} \qquad \sigma = \uplus_k \sigma_k \qquad (\Sigma;\Gamma \vdash \lceil p_k \rceil \sigma = w_k : A_k)_{k=1\ldots n}}{\Sigma;\Gamma \vdash \{w_k \mathrel{/^?} p_k : A_k \mid k = 1\ldots n\} \Rightarrow \text{SOLVED}(\sigma)}$$

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

$\boxed{P\ (x : A)}$ Replace the first application pattern p in each clause by the constraint $x \mathrel{/^?} p : A$.

$$\begin{array}{rcl} \varepsilon\ (x : A) & = & \varepsilon \\ ([E]p\ \bar{q} \hookrightarrow rhs,\ P)\ (x : A) & = & ([E \cup \{x \mathrel{/^?} p : A\}]\bar{q} \hookrightarrow rhs),\ P\ (x : A) \end{array}$$

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

$\boxed{P\ .\pi}$ Keep only clauses with copattern $.\pi$, with this copattern removed.

$$\begin{array}{rcll} \varepsilon\ .\pi & = & \varepsilon & \\ ([E].\pi\ \bar{q} \hookrightarrow rhs,\ P)\ .\pi & = & ([E]\bar{q} \hookrightarrow rhs),\ P\ .\pi & \\ ([E].\pi'\ \bar{q} \hookrightarrow rhs,\ P)\ .\pi & = & P\ .\pi & \text{if } \pi \neq \pi' \end{array}$$

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

$\boxed{\Sigma \vdash P\sigma \Rightarrow P'}$ (Σ fixed, dropped from rules.)

$$\frac{}{\varepsilon\sigma \Rightarrow \varepsilon} \qquad \frac{(v \mathrel{/^?} p : A) \in E \qquad v\sigma \mathrel{/^?} p : A\sigma \Rightarrow \bot \qquad P\sigma \Rightarrow P'}{([E]\bar{q} \hookrightarrow rhs, P)\sigma \Rightarrow P'}$$

$$\frac{E = \{w_k \mathrel{/^?} p_k : A_k \mid k = 1\ldots n\} \qquad (w_k\sigma \mathrel{/^?} p_k : A_k\sigma \Rightarrow E_i)_{k=1\ldots n} \qquad P\sigma \Rightarrow P'}{([E]\bar{q} \hookrightarrow rhs, P)\sigma \Rightarrow (\left[\bigcup_i E_i\right]\bar{q} \hookrightarrow rhs), P'}$$

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

$\boxed{\Sigma \vdash v \mathrel{/^?} p : A \Rightarrow E_\bot}$ and $\boxed{\Sigma \vdash \bar{v} \mathrel{/^?} \bar{p} : \Delta \Rightarrow E_\bot}$
(Σ fixed, dropped from rules)

$$\frac{v \longrightarrow^* \mathsf{c}\ \bar{v} \qquad A \longrightarrow^* \mathsf{D}\ \bar{u} \qquad \mathsf{constructor\ c}\ \Delta_\mathsf{c} : \mathsf{D}\ \Delta \in \Sigma \qquad \bar{v} \mathrel{/^?} \bar{p} : \Delta_\mathsf{c}[\bar{u} \,/\, \Delta] \Rightarrow E_\bot}{v \mathrel{/^?} \mathsf{c}\ \bar{p} : A \Rightarrow E_\bot}$$

$$\frac{v \longrightarrow^* \mathsf{refl} \qquad A \longrightarrow^* u \equiv_B u'}{v \mathrel{/^?} \mathsf{refl} : A \Rightarrow \{\}} \qquad \frac{v \longrightarrow^* \mathsf{c}'\ \bar{v} \qquad \mathsf{c} \neq \mathsf{c}'}{v \mathrel{/^?} \mathsf{c}\ \bar{p} : A \Rightarrow \bot}$$

$$\frac{}{v \mathrel{/^?} p : A \Rightarrow \{v \mathrel{/^?} p : A\}} \qquad \frac{}{\varepsilon \mathrel{/^?} \varepsilon : \varepsilon \Rightarrow \{\}}$$

$$\frac{v \mathrel{/^?} p : A \Rightarrow E_\bot \qquad \bar{v} \mathrel{/^?} \bar{p} : \Delta[v \,/\, x] \Rightarrow E'_\bot}{v\ \bar{v} \mathrel{/^?} p\ \bar{p} : (x : A)\Delta \Rightarrow E_\bot \cup E'_\bot}$$

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

$\boxed{\Sigma;\Gamma \vdash \emptyset : A}$ (Σ fixed, dropped from rules)

$$\frac{A \longrightarrow^* \mathsf{D}\ \bar{v} \qquad \mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell\ \mathsf{where}\ \varepsilon \in \Sigma}{\Gamma \vdash \emptyset : A}$$

$$\frac{A \longrightarrow^* u \equiv_B v \qquad \Gamma \vdash_x u =^? v : B \Rightarrow \mathrm{NO}}{\Gamma \vdash \emptyset : A}$$

Fig. 16. Rules for caseless types.

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.

COSPLIT applies when C is a record type and all the user clauses have at least one projection copattern. It constructs the case tree $\mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\}$, using $P\ .\pi_i$ to construct the branch Q_i corresponding to projection $.\pi_i$.

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 $\mathsf{record}\{\}$.

SPLITCON applies when the first clause has a constraint of the form $x\ /^?\ c_j\,\bar{p}$ and the type of x in Γ is a data type. For each constructor c_i of this data type, it constructs a pattern substitution ρ_i replacing x by c_i applied to fresh variables. It then constructs the case tree $\mathsf{case}_x\{c_1\ \hat{\Delta}'_1 \mapsto Q_1; \ldots; c_n\ \hat{\Delta}'_n \mapsto Q_n\}$, 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 $x\ /^?\ \mathsf{refl}$ and the type of x in Γ is an identity type $u \equiv_A v$. It tries to unify u with v, expecting a positive success. If unification succeeds with output (Γ'_1, ρ, τ), it constructs the case tree $\mathsf{case}_x\{\mathsf{refl} \mapsto^{\tau'} Q\}$, using $\Sigma \vdash P\rho' \Rightarrow P'$ to construct the subtree Q. Here ρ' and τ' are lifted versions of ρ and τ over the part of the context that is untouched by unification.

SPLITEMPTY applies when the first clause has a constraint of the form $x\ /^?\ \emptyset$, and the type of x is a caseless type according to $\Sigma;\Gamma \vdash \emptyset : A$. It then produces the case tree $\mathsf{case}_x\{\}$.

Remark 20 (Limitations)**.** The algorithm does not detect unreachable clauses, we left that aspect out of the formal description. Further, SPLITEMPTY may leave some user patterns uninspected, which may then be ill-typed. However, an easy check whether the whole lhs $\mathsf{f}\ \lceil\bar{q}\rceil$ is well typed as a term can rule out ill-typed patterns.

5.3 Preservation of first-match semantics

Now that we have described the elaboration algorithm from a list of clauses to a well-typed case tree, we can state and prove our main correctness theorem. We already know that elaboration always produces a well-typed case tree by construction (if it succeeds), and that well-typed case trees are type-preserving (Theorem 17) and cover all cases (Theorem 18). Now we prove that the case tree we get is the right one, i.e. that it corresponds to the definition written by the user.

To prove this theorem, we assume that the clauses we get from the user have already been scope checked, i.e. each variable in the right-hand side of a clause is bound somewhere in the patterns on the left.

Definition 21. *A partially decomposed clause* $[E]\bar{q} \hookrightarrow v$ *is* well scoped *if every free variable in* v *occurs at least once as a pattern variable in either* $\bar{q}$ *or in* p *for some constraint* $(w \mathrel{/^?} p : A) \in E$.

Theorem 22. *Let* $P = \{\bar{q}_i \hookrightarrow rhs_i \mid i = 1 \ldots n\}$ *be a problem consisting of well-scoped clauses such that* $\Sigma_0 \vdash P \mid f := Q : C \rightsquigarrow \Sigma$ *and let* $\Sigma; \Gamma \mid f : C \vdash \bar{e} : B$ *be eliminations. Let* i *be the index of the first matching clause, i.e.:*

- $[\bar{e} \mathbin{/} \bar{q}_j] \Rightarrow \bot$ *for* $j = 1 \ldots i - 1$.
- $[\bar{e} \mathbin{/} \bar{q}_i] \Rightarrow \sigma$.

Then $rhs_i = u_i$ *is not* impossible *and* $\Sigma \vdash Q[\,]\, f\bar{e} \longrightarrow u_i\sigma$.

For the proof, we first need two basic properties of the auxiliary judgement $\Sigma \vdash v \mathrel{/^?} p : A \Rightarrow E$.

Lemma 23. *If* $\Sigma \vdash v \mathrel{/^?} p : A \Rightarrow E$ *where* $E = \{w_k \mathrel{/^?} p_k : B_k \mid k = 1 \ldots l\}$, *then for any substitution* σ *we also have* $\Sigma \vdash v\sigma \mathrel{/^?} p : A\sigma \Rightarrow E'$ *where* $E' = \{w_k\sigma \mathrel{/^?} p_k : B_k\sigma \mid k = 1 \ldots l\}$.

Proof This follows directly from the rules of simplification (Figure 15). □

Lemma 24. *Let* σ *be a substitution and suppose* $\Sigma \vdash v \mathrel{/^?} p : A \Rightarrow E$. *Then the following hold*

- $[v\sigma \mathbin{/} p] \Rightarrow \sigma'$ *if and only if for each* $(w_k \mathrel{/^?} p_k : A_k) \in E$, *we have* $[w_k\sigma \mathbin{/} p_k] \Rightarrow \sigma_k$, *and* $\sigma' = \biguplus_k \sigma_k$.
- $[v\sigma \mathbin{/} p] \Rightarrow \bot$ *if and only if for some* $(w_k \mathrel{/^?} p_k : A_k) \in E$, *we have* $[w_k\sigma \mathbin{/} p_k] \Rightarrow \bot$.

Proof This follows directly from the rules of matching (Figure 6) and simplification (Figure 15). □

The following lemma is the main component of the proof. It generalizes the statement of Theorem 22 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 $\bar{q} = \varepsilon$ and $E_i = \{\}$ for $i = 1 \ldots n$.

Lemma 25. *Let* $P = \{[E_i]\bar{q}_i \hookrightarrow rhs_i \mid i = 1 \ldots n\}$ *be a list of well-scoped partially decomposed clauses such that* $\Sigma_0; \Gamma_0 \vdash P \mid f\,\bar{q} := Q : C \rightsquigarrow \Sigma$, *and suppose* $\Gamma \vdash \sigma_0 : \Gamma_0$ *and* $\Sigma; \Gamma \mid f\bar{q}\sigma_0 : C\sigma_0 \vdash \bar{e} : B$. *If there is an index* i *such that:*

- *For each* $j = 1 \ldots i - 1$ *and each constraint* $(w_k \mathrel{/^?} p_k : A_k) \in E_j$, *either* $[w_k\sigma_0 \mathbin{/} p_k] \Rightarrow \theta_{jk}$ *or* $[w_k\sigma_0 \mathbin{/} p_k] \Rightarrow \bot$.
- *For each* $j = 1 \ldots i - 1$, *either* $[\bar{e} \mathbin{/} \bar{q}_j] \Rightarrow \theta_{j0}$ *or* $[\bar{e} \mathbin{/} \bar{q}_j] \Rightarrow \bot$.

- *For each* $j = 1 \ldots i-1$, *either* $[w_k\sigma_0 \,/\, p_k] \Rightarrow \bot$ *for some constraint* $(w_k \,/^?\, p_k : A_k) \in E_j$ *or* $[\bar{e} \,/\, \bar{q}_j] \Rightarrow \bot$.
- *For each* $(w_k \,/^?\, p_k : A_k) \in E_i$, *we have* $[w_k\sigma_0 \,/\, p_k] \Rightarrow \theta_k$.
- $[\bar{e} \,/\, \bar{q}_i] \Rightarrow \theta_0$.

Then $rhs_i = v_i$ *is not* impossible *and* $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow v_i\theta$ *where* $\theta = \theta_0 \uplus (\biguplus_k \theta_k)$.

Proof By induction on the derivation of $\Sigma_0; \Gamma_0 \vdash P \mid \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma$:

- For the DONE rule where $Q = v_1\sigma$ and $\Sigma_0 = \Sigma$, we have $\bar{q}_1 = \varepsilon$ and $rhs_1 = v_1$ (i.e. rhs_1 is not impossible). We also get that $\sigma = \biguplus_k \sigma_k$ is a substitution such that $\Sigma; \Gamma_0 \vdash v_1\sigma : C$ and $[w_k \,/\, p_k] \Rightarrow \sigma_k$ and $\Sigma; \Gamma_0 \vdash \lceil p_k \rceil\sigma = w_k : A_k$ for each $(w_k \,/^?\, p_k : A_k) \in E_1$. Because $Q = v_1\sigma$, we have $\Sigma \vdash Q\sigma_0\,\varepsilon \longrightarrow v_1\sigma\sigma_0$, so what is left to prove is that $v_1\sigma\sigma_0 = v_i\theta$ (syntactically). First, we show that $i = 1$, i.e. the first clause matches. Since $[w_k \,/\, p_k] \Rightarrow \sigma_k$, we cannot have $[w_k\sigma_0 \,/\, p_k] \Rightarrow \bot$, and since $\bar{q}_1 = \varepsilon$, we also cannot have $[\bar{e} \,/\, \bar{q}_1] \Rightarrow \bot$. The only remaining possibility is that i is 1. This means we have $[w_k\sigma_0 \,/\, p_k] \Rightarrow \theta_k$ for each $(w_k \,/^?\, p_k : A_k) \in E_1$ and $[\bar{e} \,/\, \bar{q}_1] \Rightarrow \theta_0$. Since $\bar{q}_1 = \varepsilon$, we also have $\bar{e} = \varepsilon$ and $\theta_0 = [\,]$. To finish this case, we show that $\sigma\sigma_0 = (\biguplus_k \sigma_k)\sigma_0$ and $\theta = \biguplus_k \theta_k$ coincide on all free variables in v. Since the clause $[E_1] \hookrightarrow v_1$ is well scoped, for each free variable x in v_1, there is at least one constraint $(w_k \,/^?\, p_k : A_k) \in E_1$ such that x is a pattern variable of p_k. Since we have both $[w_k \,/\, p_k] \Rightarrow \sigma_k$ and $[w_k\sigma_0 \,/\, p_k] \Rightarrow \theta_k$, we have $x\sigma_k\sigma_0 = x\theta_k$. This holds for any free variable x in v_1, so we have $v_1\sigma\sigma_0 = v_1\theta$, finishing the proof for the base case.
- For the INTRO rule, we have $Q = \lambda x.\, Q'$ where $\Sigma_0 \vdash C \longrightarrow^* (x : A) \to B$ and $\bar{q}_i = p_i\,\bar{q}'_i$ for $i = 1 \ldots n$. We also know that $\Sigma_0; \Gamma_0(x : A) \vdash P' \mid \mathsf{f}\,\bar{q}\,x := Q' : B \rightsquigarrow \Sigma$ where $P' = P\,(x : A)$. Since we have either $[\bar{e} \,/\, p_1\,\bar{q}'_1] \Rightarrow \bot$ or $[\bar{e} \,/\, p_1\,\bar{q}'_1] \Rightarrow \theta_0$, we have $\bar{e} = t\,\bar{e}'$ for some term t. Now we apply the induction hypothesis to show that $rhs_i = v_i$ is not impossible and $\Sigma \vdash Q'(\sigma_0 \uplus [t \,/\, x])\,\bar{e}' \longrightarrow v_i\theta$, hence also $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow v_i\theta$.
- For the COSPLIT rule where $Q = \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_n \mapsto Q_n\}$, we know that $\Sigma_0 \vdash C \longrightarrow^* \mathsf{R}\,\bar{v}$ and $\bar{q}_1 = .\pi_\alpha\,\bar{q}'_1$ where $\mathsf{projection}\,x : \mathsf{R}\,\Delta \vdash .\pi_\alpha : A_\alpha \in \Sigma_0$. We know that either $[\bar{e} \,/\, .\pi_\alpha\,\bar{q}'_1] \Rightarrow \theta_{10}$ or $[\bar{e} \,/\, .\pi_\alpha\,\bar{q}'_1] \Rightarrow \bot$, so $\bar{e} = .\pi_\beta\,\bar{e}'$ for some $\mathsf{projection}\,x : \mathsf{R}\,\Delta \vdash .\pi_\beta : A_\beta \in \Sigma_0$. We then have $\Sigma_{\beta-1}; \Gamma \vdash P\,.\pi_\beta \mid \mathsf{f}\,\bar{q}\,.\pi_\beta := Q_\beta : A_\beta[\bar{v} \,/\, \Delta; u \,/\, x] \rightsquigarrow \Sigma_\beta$. By induction, we have that rhs_i is not impossible and $\Sigma_\beta \vdash Q_\beta\sigma_0\,\bar{e}' \longrightarrow v\theta$, hence also $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow v\theta$.
- For the COSPLITEMPTY rule, we have $\bar{q}_1 = \emptyset\,\bar{q}'_1$. Since there are no rules for $[.\pi \,/\, \emptyset] \Rightarrow \theta_\bot$, this case is impossible.
- For the SPLITCON rule, we know that $Q = \mathsf{case}_x\{c_1\,\hat{\Delta}'_1 \mapsto Q_1; \ldots; c_n\,\hat{\Delta}'_n \mapsto Q_n\}$ where $n \geq 1$, $\Gamma = \Gamma_1(x : A)\Gamma_2$ and $\Sigma_0 \vdash A \longrightarrow^* \mathsf{D}\,\bar{v}$. Since $(x \,/^?\, c_\alpha\,\bar{p} : A) \in E_1$, we either have $[x\sigma_0 \,/\, c_\alpha\,\bar{p}] \Rightarrow \theta_{1k}$ or $[x\sigma_0 \,/\, c_\alpha\,\bar{p}] \Rightarrow \bot$ (this is the case both if $i = 1$ and if $i > 1$). In either case, we have $x\sigma_0 = c_\beta\,\bar{u}$ for some $\mathsf{constructor}\,c_\beta\,\Delta_\beta : \mathsf{D}\,\Delta \in \Sigma_0$. Let $\Delta'_\beta = \Delta_\beta[\bar{v} \,/\, \Delta]$ and $\rho_\beta = [c_\beta\,\hat{\Delta}'_\beta \,/\, x]$, then we have $\Sigma \vdash P\rho_\beta \Rightarrow P_\beta$ and $\Sigma_{\beta-1}; \Gamma_1\Delta'_\beta\Gamma_2\rho_\beta \vdash P_\beta \mid \mathsf{f}\,\bar{q}\rho_\beta := Q_\beta : C\rho_\beta \rightsquigarrow \Sigma_\beta$. We now apply the induction hypothesis to get that $rhs_i = v_i$ is not impossible and $\Sigma_\beta \vdash Q_\beta(\sigma_0 \uplus [\bar{u} \,/\, \Delta'_\beta\sigma_0])\,\bar{e} \longrightarrow v_i\theta$, hence also $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow v_i\theta$.

- For the SplitEq rule where $Q = \mathsf{case}_x\{\mathsf{refl} \mapsto^{\tau} Q\}'$, we know that $\Gamma = \Gamma_1(x : A)\Gamma_2$ and $\Sigma_0 \vdash A \longrightarrow^* u \equiv_A v$. Since $(x \,/^?\, \mathsf{refl} : A) \in E_1$, we either have $[x\sigma_0 \,/\, \mathsf{refl}] \Rightarrow \theta_{1k}$ or $[x\sigma_0 \,/\, \mathsf{refl}] \Rightarrow \bot$. However, the latter case is impossible since refl is the only constructor of the identity type, so we have $x\sigma_0 = \mathsf{refl}$ and $\theta_{1k} = [\,]$. We moreover have $\Sigma_0; \Gamma_1 \vdash_x u =^? v : B \Rightarrow \mathrm{YES}(\Gamma_1', \rho, \tau)$ and $\Sigma_0; \Gamma_1'\Gamma_2\rho \vdash P' \mid \mathsf{f}\,\bar{q}\rho' := Q' : C\rho' \rightsquigarrow \Sigma$ where $\rho' = \rho \uplus \mathbb{1}_{\Gamma_2}$ and $\Sigma_0 \vdash P\rho \Rightarrow P'$. By induction (and using Definition 8 to show that $\rho'; \tau; \sigma_0 = \sigma_0$), we get that $rhs_i = v_i$ is not impossible and $\Sigma \vdash Q'(\tau; \sigma_0)\,\bar{e} \longrightarrow \theta$, hence also $\Sigma \vdash Q\sigma_0\,\bar{e} \longrightarrow \theta$.
- For the SplitEmpty rule, we know that $Q = \mathsf{case}_x\{\}$ and $(x \,/^?\, \emptyset : A) \in E_1$ where $\Sigma_0; \Gamma \vdash \emptyset : A$. We either have $[x\sigma_0 \,/\, \emptyset] \Rightarrow \theta_x$ or $[x\sigma_0 \,/\, \emptyset] \Rightarrow \bot$. However, there are no rules for $[v \,/\, \emptyset] \Rightarrow \theta_\bot$ so this case is impossible.

□

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 Section 2.4).

McBride & McKinna (2004) present an alternative syntax for dependent pattern matching where the user specifies the order of case splitting explicitly. This gives more control to the user, but obfuscates the equational presentation of the clauses. It would be interesting to see how to extend our syntax with optional splitting annotations for when more control is desired.

The translation from a case tree to primitive data type 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^{ω}(Abel & 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 (co)pattern 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. Likewise, Laforgue and Régis-Gianas (2017) extend OCaml with copatterns, but they do not allow the types of record fields to depend on each other.

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 Conclusion

In this paper, we give a description of an elaboration algorithm for definitions by dependent copattern 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 main motivation for writing this paper was to get a clear idea of how to implement a good elaboration algorithm for dependent (co)pattern matching that works for a full-featured dependently typed language such as Agda. Working out the theory allowed us to uncover several potential issues and edge cases in the implementation. For instance, while working on the proof of Theorem 22, 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, 2018b). Removing the shortcut rule from the definition of matching removed this behavioural divergence mismatch. The complete formalization of the elaboration algorithm in this paper lets us continue to work on the implementation with confidence.

Agda also has a number of features that are not described in this paper, such as non-recursive record types with η equality and general indexed data types (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.

Acknowledgments

The authors acknowledge support by the Swedish Research Council (Vetenskapsrådet) under Grant No. 621-2014-4864 *Termination Certificates for Dependently-Typed*

Programs and Proofs via Refinement Types. Our research group is part of the EU Cost Action CA15123 The European research network on types for programming and verification (EUTypes). We thank the anonymous referees of both the conference version and the current version of this paper for their comments that helped improving the presentation.

Conflicts of Interest

None

References

Abel, A. & Pientka, B. (2013) Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *Proceedings of the Eighteenth ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA, September 25–27, 2013*, Morrisett, G. & Uustalu, T. (eds). ACM Press, pp. 185–196. Available at: http://doi.acm.org/10.1145/2500365.2500591

Abel, A., Pientka, B., Thibodeau, D. & Setzer, A. (2013) Copatterns: Programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23–25, 2013*, Giacobazzi, R. & Cousot, R. (eds). ACM Press, pp. 27–38. Available at: http://dl.acm.org/citation.cfm?id=2429069

Agda development team. (2017) Agda 2.5.3 documentation. Available at: http://agda.readthedocs.io/en/v2.5.3/

Agda issue. (2017a) Disambiguation of type based on pattern leads to non-unique meta solution. Available at: https://github.com/agda/agda/issues/2834 (on the Agda bug tracker).

Agda issue. (2017b) Record constructor is accepted, record pattern is not. Available at: https://github.com/agda/agda/issues/2850 (on the Agda bug tracker).

Agda issue (2017c). Panic: Unbound variable. Available at: https://github.com/agda/agda/issues/2856 (on the Agda bug tracker).

Agda issue. (2017d) Internal error in src/full/agda/typechecking/coverage/match.hs:312. Available at: https://github.com/agda/agda/issues/2874. (on the Agda bug tracker).

Agda issue. (2018a) Unifier throws away pattern. Available at: https://github.com/agda/agda/issues/2896. (on the Agda bug tracker).

Agda issue. (2018b) Mismatch between order of matching in clauses and case tree; subject reduction broken. Available at: https://github.com/agda/agda/issues/2964. (on the Agda bug tracker).

Andreoli, J.-M. (1992) Logic programming with focusing proofs in linear logic. *J. Logic Comput.* **2**(3), 297–347. Available at: https://doi.org/10.1093/logcom/2.3.297.

Augustsson, L. (1985) Compiling pattern matching. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, Proceedings September 16–19, 1985*, vol. 201. Lecture Notes in Computer Science. Springer, pp. 368–381. Available at: https://doi.org/10.1007/3-540-15975-4_48.

Blanchette, J. C., Bouzy, A., Lochbihler, A., Popescu, A. & Traytel, D. (2017) Friends with benefits – implementing corecursion in foundational proof assistants. In *Programming Languages and Systems – 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*, Yang, H. (ed), vol. 10201. Lecture Notes in Computer Science. Springer, pp. 111–140. Available at: https://doi.org/10.1007/978-3-662-54434-1_5

Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Functional Program.* **23**(5), 552–593. Available at: http://dx.doi.org/10.1017/S095679681300018X

Brady, E., McBride, C. & McKinna, J. (2003) Inductive families need not store their indices. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30–May 4, 2003, Revised Selected Papers*, Berardi, S., Coppo, M. & Damiani, F. (eds), vol. 3085. Lecture Notes in Computer Science. Springer, pp. 115–129. Available at: https://doi.org/10.1007/978-3-540-24849-1_8

Cardelli, L. (1984) Compiling a functional language. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, August 5–8, 1984, Austin, Texas, USA*. ACM Press, pp. 208–217. Available at: http://lucacardelli.name/Papers/CompilingML.A4.pdf

Cockx, J. (2017) Dependent pattern matching and proof-relevant unification. PhD thesis, KU Leuven, 2017.

Cockx, J. & Abel, A. (2018) Elaborating dependent (co)pattern matching. *PACMPL*, **2**(ICFP), 75:1–75:30. Available at: https://doi.org/10.1145/3236770.

Cockx, J., Devriese, D. & Piessens, F. (2016) Unifiers as equivalences: Proof-relevant unification of dependently typed data. In Garrigue *et al*. (2016), pp. 270–283. Available at: http://doi.acm.org/10.1145/2951913.2951917.

Coquand, T. (1992) Pattern matching with dependent types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, Nordström, B., Pettersson, K. & Plotkin, G. (eds). Chalmers University of Technology, pp. 71–83, 1992. Available at: http://www.cse.chalmers.se/~coquand/pattern.ps

de Bruijn, N. G. (1991) Telescopic mappings in typed lambda calculus. *Inf. Comput.* **91**(2), 189–204. Available at: https://doi.org/10.1016/0890-5401(91)90066-B

de Moura, L. M., Kong, S., Avigad, J., van Doorn, F. & von Raumer, J. (2015) The Lean theorem prover (system description). In *Automated Deduction – CADE-25 – 25th International Conference on Automated Deduction, Berlin, Germany, August 1–7, 2015, Proceedings*, Felty, A. P. & Middeldorp, A. (eds), vol. 9195. Lecture Notes in Computer Science. Springer, pp. 378–388 Available at: https://doi.org/10.1007/978-3-319-21401-6_26

Garrigue, J., Keller, G. & Sumii, E. (eds). *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016*. ACM Press. Available at: http://doi.acm.org/10.1145/2951913

Goguen, H., McBride, C. & McKinna, J. (2006) Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Futatsugi, K., Jouannaud, J.-P. & Meseguer, J. (eds), vol. 4060. Lecture Notes in Computer Science. Springer, pp. 521–540. Available at: https://doi.org/10.1007/11780274_27

INRIA. (2017) *The Coq Proof Assistant Reference Manual*. INRIA, version 8.7 edition, Available at: http://coq.inria.fr/

Krishnaswami, N. R. (2009) Focusing on pattern matching. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21–23, 2009*, Shao, Z. & Pierce, B. C. (eds). ACM Press, pp. 366–378. Available at: http://doi.acm.org/10.1145/1480881.1480927

Laforgue, P. & Régis-Gianas, Y. (2017) Copattern matching and first-class observations in ocaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09–11, 2017*, Vanhoof, W. & Pientka, B. (eds). ACM, pp. 97–108. ISBN 978-1-4503-5291-8. Available at: https://doi.org/10.1145/3131851.3131869

Licata, D. R., Zeilberger, N. & Harper, R. (2008) Focusing on binding and computation. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24–27 June 2008, Pittsburgh, PA, USA*, Pfenning, F. (ed). IEEE Computer Society Press, pp. 241–252. Available at: https://doi.org/10.1109/LICS.2008.48

Maranget, L. (1992) Compiling lazy pattern matching. In *LISP and Functional Programming*, pp. 21–31. Available at: http://doi.acm.org/10.1145/141471.141499

McBride, C. (2000) Dependently typed functional programs and their proofs. PhD thesis, University of Edinburgh.

McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1), 69–111. Available at: https://doi.org/10.1017/S0956796803004829.

Norell, U. (2007) Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology.

Oury, N. (2007) Pattern matching coverage checking with dependent types using set approximations. In *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, Stump, A. & Xi, H. (eds), pp. 47–56. ACM Press. Available at: http://doi.acm.org/10.1145/1292597.1292606

Pollack, R. (1998) How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press. Available at: http://www.brics.dk/RS/97/18/BRICS-RS-97-18.pdf

Setzer, A., Abel, A., Pientka, B. & Thibodeau, D. (2014) Unnesting of copatterns. In *Rewriting and Typed Lambda Calculi – Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, Dowek, G. (ed), vol. 8560. Lecture Notes in Computer Science. Springer, pp. 31–45. Available at: http://dx.doi.org/10.1007/978-3-319-08918-8_3

Sozeau, M. (2010) Equations: A dependent pattern-matching compiler. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11–14, 2010. Proceedings*, Kaufmann, M. & Paulson, L. C. (eds), vol. 6172. Lecture Notes in Computer Science. Springer, pp. 419–434. Available at: https://doi.org/10.1007/978-3-642-14052-5_29

Thibodeau, D., Cave, A. & Pientka, B. (2016) Indexed codata types. In Garrigue *et al*. (2016), pp. 351–363. Available at: http://doi.acm.org/10.1145/2951913.2951929

Zeilberger, N. (2008) Focusing and higher-order abstract syntax. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7–12, 2008*, Necula, G. C. & Wadler, P. (eds) pp. 359–369. ACM Press. Available at: http://doi.acm.org/10.1145/1328438.1328482

Zeilberger, N. (2009) The Logical Basis of Evaluation Order and Pattern-Matching. PhD thesis, Carnegie Mellon University. Available at: http://software.imdea.org/~noam.zeilberger/thesis.pdf

A. Inference Rules

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \varepsilon} \qquad \frac{\Gamma \vdash A \qquad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma(x : A)}$$

Fig. A.1. The typing rules for valid contexts.

$\boxed{\Gamma \vdash u = v : A}$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash u = u : A} \qquad \frac{\Gamma \vdash u_1 = u_2 : A}{\Gamma \vdash u_2 = u_1 : A} \qquad \frac{\Gamma \vdash u_1 = u_2 : A \qquad \Gamma \vdash u_2 = u_3 : A}{\Gamma \vdash u_1 = u_3 : A}$$

$$\frac{\Gamma \vdash u_1 = u_2 : A_1 \qquad \Gamma \vdash A_1 = A_2}{\Gamma \vdash u_1 = u_2 : A_2} \qquad \frac{\Gamma \vdash A_1 = A_2 : \mathsf{Set}_\ell \qquad \Gamma(x : A_1) \vdash B_1 = B_2 : \mathsf{Set}_{\ell'}}{\Gamma \vdash (x : A_1) \to B_1 = (x : A_2) \to B_2 : \mathsf{Set}_{\max(\ell,\ell')}}$$

$$\frac{\vdash \Gamma \qquad x : A \in \Gamma \qquad \Gamma \mid x : A \vdash \bar{e}_1 = \bar{e}_2 : B}{\Gamma \vdash x\ \bar{e}_1 = x\ \bar{e}_2 : B}$$

$$\frac{\Gamma \vdash A_1 = A_2 : \mathsf{Set}_\ell \qquad \Gamma \vdash u_1 = u_2 : A_1 \qquad \Gamma \vdash v_1 = v_2 : A_1}{\Gamma \vdash (u_1 \equiv_{A_1} v_1) = (u_2 \equiv_{A_2} v_2) : \mathsf{Set}_\ell}$$

$$\frac{\mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_\ell \in \Sigma \qquad \Gamma \vdash \bar{u}_1 = \bar{u}_2 : \Delta}{\Gamma \vdash \mathsf{D}\ \bar{u}_1 = \mathsf{D}\ \bar{u}_2 : \mathsf{Set}_\ell} \qquad \frac{\mathsf{record}\ \mathsf{R}\ \Delta : \mathsf{Set}_\ell \in \Sigma \qquad \Gamma \vdash \bar{u}_1 = \bar{u}_2 : \Delta}{\Gamma \vdash \mathsf{R}\ \bar{u}_1 = \mathsf{R}\ \bar{u}_2 : \mathsf{Set}_\ell}$$

$$\frac{\mathsf{constructor}\ \mathsf{c}\ \Delta_\mathsf{c} : \mathsf{D}\ \Delta \in \Sigma \qquad \Gamma \vdash \bar{u} : \Delta \qquad \Gamma \vdash \bar{v}_1 = \bar{v}_2 : \Delta_\mathsf{c}[\bar{u}/\Delta]}{\Gamma \vdash \mathsf{c}\ \bar{v}_1 = \mathsf{c}\ \bar{v}_2 : \mathsf{D}\ \bar{u}}$$

$$\frac{\mathsf{definition}\ \mathsf{f} : A \in \Sigma \qquad \Gamma \mid \mathsf{f} : A \vdash \bar{e}_1 = \bar{e}_2 : B}{\Gamma \vdash \mathsf{f}\ \bar{e}_1 = \mathsf{f}\ \bar{e}_2 : B} \qquad \frac{\mathsf{clause}\ \Delta \vdash \mathsf{f}\ \bar{q} \hookrightarrow v : B \in \Sigma \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash \mathsf{f}\ \lceil \bar{q} \rceil \sigma = v\sigma : B\sigma}$$

Fig. A.2. The conversion rules for terms.

$\boxed{\Gamma \mid u : A \vdash \bar{e}_1 = \bar{e}_2 : B}$

$$\frac{}{\Gamma \mid u : A \vdash \varepsilon = \varepsilon : A} \qquad \frac{\Gamma \vdash v_1 = v_2 : A \qquad \Gamma \mid u\ v_1 : B[v_1/x] \vdash \bar{e}_1 = \bar{e}_2 : C}{\Gamma \mid u : (x : A) \to B \vdash v_1\ \bar{e}_1 = v_2\ \bar{e}_2 : C}$$

$$\frac{\mathsf{projection}\ x : \mathsf{R}\ \Delta \vdash .\pi : A \in \Sigma \qquad \Gamma \mid u\ .\pi : A[\bar{v}/\Delta, u/x] \vdash \bar{e}_1 = \bar{e}_2 : C}{\Gamma \mid u : \mathsf{R}\ \bar{v} \vdash .\pi\ \bar{e}_1 = .\pi\ \bar{e}_2 : C}$$

$$\frac{\Gamma \mid u : A \vdash \bar{e}_1 = \bar{e}_2 : B \qquad \Gamma \vdash A = A' \qquad \Gamma \vdash B = B'}{\Gamma \mid u : A' \vdash \bar{e}_1 = \bar{e}_2 : B'}$$

Fig. A.3. The conversion rules for eliminations.

$\boxed{\Gamma \vdash \bar{u} = \bar{v} : \Delta}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \varepsilon = \varepsilon : \varepsilon} \qquad \frac{\Gamma \vdash u_1 = u_2 : A \qquad \Gamma \vdash \bar{u}_1 = \bar{u}_2 : \Delta[u_1/x]}{\Gamma \vdash u_1\ \bar{u}_1 = u_2\ \bar{u}_2 : (x : A)\Delta}$$

Fig. A.4. The conversion rules for lists of terms.