Overlapping and Order-Independent Patterns
Definitional Equality for All

Jesper Cockx, Frank Piessens, and Dominique Devriese

DistriNet, KU Leuven, Belgium
`firstname.lastname@cs.kuleuven.be`

Abstract. Dependent pattern matching is a safe and efficient way to write programs and proofs in dependently typed languages. Current languages with dependent pattern matching treat overlapping patterns on a first-match basis, hence the order of the patterns can matter. Perhaps surprisingly, this order-dependence can even occur when the patterns do not overlap. To fix this confusing behavior, we developed a new semantics of pattern matching which treats all clauses as definitional equalities, even when the patterns overlap. A confluence check guarantees correctness in the presence of overlapping patterns. Our new semantics has two advantages. Firstly, it removes the order-dependence and thus makes the meaning of definitions clearer. Secondly, it allows the extension of existing definitions with new (consistent) evaluation rules. Unfortunately it also makes pattern matching harder to understand theoretically, but we give a theorem that helps to bridge this gap. An experimental implementation in Agda shows that our approach is feasible in practice too.

Keywords: Type theory, dependent pattern matching, overlapping patterns, confluence, Agda

1 Introduction

Pattern matching is a mechanism to write programs by case distinction and recursion. Definitions by pattern matching are given by a set of equalities called *clauses*, for example:

$$\begin{array}{llll} \texttt{plus} : \texttt{Nat} \to & \texttt{Nat} \to \texttt{Nat} & & \\ \texttt{plus}\ \ \texttt{zero} & n & = n & \\ \texttt{plus}\ \ (\texttt{suc}\ m) & n & = \texttt{suc}\ (\texttt{plus}\ m\ n) & \end{array} \tag{1}$$

If the patterns of the clauses of a definition overlap, it is customary to choose the first clause that gives a match. This is the *first-match semantics* of pattern matching. For example, in the following definition the last clause cannot hold as a definitional equality but holds only when the first two clauses don't match:

$$\begin{array}{llll} \texttt{equal} : \texttt{Nat} \to & \texttt{Nat} & \to \texttt{Bool} \\ \texttt{equal}\ \ \texttt{zero} & \texttt{zero} & = \texttt{true} \\ \texttt{equal}\ \ (\texttt{suc}\ m) & (\texttt{suc}\ n) & = \texttt{equal}\ m\ n \\ \texttt{equal}\ \ m & n & = \texttt{false} \end{array} \tag{2}$$

In a language with dependent types, pattern matching allows us to write not just programs, but also proofs. For example, the following is a proof that $\texttt{plus}\ m\ \texttt{zero} \equiv m$ for all $m : \texttt{Nat}$:[1]

$$\begin{array}{lll}
\texttt{lemma} : (m : \texttt{Nat}) & \to & \texttt{plus}\ m\ \texttt{zero} \equiv m \\
\texttt{lemma}\ \ \texttt{zero} & = & \texttt{refl} \\
\texttt{lemma}\ \ (\texttt{suc}\ m) & = & \texttt{cong}\ \texttt{suc}\ (\texttt{lemma}\ m)
\end{array} \tag{3}$$

If we are not careful, it is very well possible to give incorrect proofs by pattern matching. For example, a case analysis might be incomplete, or a recursive proof might become infinitely large when we expand it. This leads to an inconsistent logic. Hence certain restrictions are put on definitions by pattern matching to ensure totality [Coq92]. These restrictions allow us to translate definitions by pattern matching to type theory with only the theoretically simpler *eliminators* plus the K axiom [GMM06]. This ensures that definitions by pattern matching are correct with respect to the core theory, but also limits the expressiveness of the language.

In order to guarantee completeness, it is required that patterns must form a *covering*, i.e. arise as the patterns at the leaves of a *case tree*. An example of a case tree for a function $\texttt{half} : \texttt{Nat} \to \texttt{Nat}$ is given in Fig. 1. This case tree shows that the patterns $\texttt{zero}$, $\texttt{suc zero}$, and $\texttt{suc}\ (\texttt{suc}\ k)$ together form a covering, ensuring completeness of functions that use these patterns.

$$\begin{array}{lll}
\texttt{half} : \texttt{Nat} & \to & \texttt{Nat} \\
\texttt{half}\ \ \texttt{zero} & = & \texttt{zero} \\
\texttt{half}\ \ (\texttt{suc}\ \texttt{zero}) & = & \texttt{zero} \\
\texttt{half}\ \ (\texttt{suc}\ (\texttt{suc}\ n)) & = & \texttt{suc}\ (\texttt{half}\ n)
\end{array}
\qquad
\underline{n} \begin{cases} \texttt{zero} \mapsto \texttt{zero} \\ (\texttt{suc}\ \underline{m}) \begin{cases} \texttt{suc}\ \texttt{zero} \mapsto \texttt{zero} \\ \texttt{suc}\ (\texttt{suc}\ k) \mapsto \texttt{suc}\ (\texttt{half}\ k) \end{cases} \end{cases}$$

Fig. 1. Case trees such as the one on the right are used to check completeness. In each internal node, one variable is chosen and replaced by all possible constructors of its type applied to fresh variables.

Some languages with dependent pattern matching (such as Agda [Nor07]) allow more general pattern sets, but translate them to a covering internally. In this translation, overlapping patterns are treated on a first-match basis, hence the result of the translation depends on the order of the clauses.

Perhaps surprisingly, this order-dependence occurs even when the patterns do not overlap. For example, if we define disjunction on booleans as in [Ab12]:

$$\begin{array}{llll}
\texttt{or} : \texttt{Bool} & \to \texttt{Bool} & \to & \texttt{Bool} \\
\texttt{or}\ \ \texttt{false} & \texttt{false} & = & \texttt{false} \\
\texttt{or}\ \ \texttt{true} & \texttt{false} & = & \texttt{true} \\
\texttt{or}\ \ x & \texttt{true} & = & \texttt{true}
\end{array} \tag{4}$$

[1] The *identity type* $a \equiv b$ expresses equality of two terms $a, b : A$. Here $\texttt{refl}$ is a proof that $m \equiv m$ and $\texttt{cong}\ f\ p$ is a proof that $f\ x \equiv f\ y$ if p is a proof that $x \equiv y$.

then it does not satisfy the definitional equality[2] `or` x `true` = `true`, while this is the case if the last clause is given first instead, leading to unexpected results for an inexperienced user. This is a sign of bad abstraction.

The goal of this paper is to make dependent pattern matching more amenable to equational reasoning. We do this by interpreting each clause directly as a definitional equality, even when the patterns overlap. In particular, our interpretation does not depend on the order of the patterns. This also allows us to give definitions with overlapping patterns, which can be used to extend a function with extra evaluation rules. For example, we allow the following definition:

```
plus : Nat →  Nat     → Nat
plus  zero    y       = y
plus  (suc x) y       = suc (plus x y)        (5)
plus  x       zero    = x
plus  x       (suc y) = suc (plus x y)
```

While all the examples in this introduction only use simple types, our approach is general enough to cope with *inaccessible patterns*, which are specific to dependent pattern matching. Section 6 includes two examples of dependent functions with overlapping patterns.

By making all clauses hold as definitional equalities, definitions by pattern matching feel more like mathematical definitions, rather than sequential program instructions. However, we lose the ability to translate pattern matching to the use of eliminators, making it more complex to understand theoretically.

Contributions

– We present an extended form of dependent pattern matching that allows patterns that do not necessarily form a covering (e.g. they might overlap), while treating all clauses as definitional equalities.
– We give a generalized criterion for completeness of overlapping patterns.
– We describe a simple criterion that can be used to check the confluence of definitions with overlapping patterns.
– We verify the feasibility of our approach by extending the Agda language, and give some simple examples that show how overlapping patterns can be used to add extra computation rules to existing functions.
– We formulate and prove a theoretical result that gives for every definition of a function f with overlapping patterns another definition of a function f' of which the patterns form a covering such that f' is extensionally equal to f.

Outlook In Sect. 2, we give our notations and conventions for this paper. In Sect. 3, we describe the three problems with dependent pattern matching in current languages that we try to solve. In Sect. 4, we give a general description of our extended form of dependent pattern matching. In Sect. 5, we describe how

[2] Two terms are called *definitionally equal* if they have the same normal form.

the correctness of these extended definitions by pattern matching can be checked. In Sect. 6, we give some examples of how our extended form of pattern matching can be used. In Sect. 7, we give a theoretical result that says that each definition that uses our extension is extensionally equal to a classical one.

2 Conventions and Terminology

Type Theory. As our version of type theory, we use Luo's Unified Theory of Dependent Types (UTT) with dependent products, inductive families, and universes [Luo94]. We omit the meta-level logical framework and the impredicative universe of propositions because they are not needed for our current work. The formal rules of the version of UTT we use are summarized in Fig. 2.

$$\frac{}{\epsilon\ \mathbf{valid}}\ (\text{Ctx-empty}) \qquad \frac{\Gamma \vdash A : Set_i \qquad x \notin FV(\Gamma)}{\Gamma(x : A)\ \mathbf{valid}}\ (\text{Ctx-ext})$$

$$\frac{\Gamma\ \mathbf{valid} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma \vdash t : A_1 \qquad \Gamma \vdash A_1 = A_2 : Set_i}{\Gamma \vdash t : A_2}\ (=\text{Ty})$$

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash \epsilon : \epsilon}\ (\text{List-empty}) \qquad \frac{\Gamma \vdash \bar{t} : \Delta \qquad \Gamma \vdash t : A[\Delta \mapsto \bar{t}]}{\Gamma \vdash \bar{t}\, t : \Delta(x : A)}\ (\text{List-ext}) \quad + \text{equality rule}$$

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash Set_i : Set_{i+1}}\ (\text{Set}) \qquad \frac{\Gamma \vdash A : Set_i \qquad \Gamma(x : A) \vdash B : Set_j}{\Gamma \vdash (x : A) \to B : Set_{\max(i,j)}}\ (\Pi) \quad + \text{equality rule}$$

$$\frac{\Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda(x : A).\, t : (x : A) \to B}\ (\lambda) \qquad + \text{equality rule}$$

$$\frac{\Gamma(x : A) \vdash B : Set_i \qquad \Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B[x \mapsto t]}\ (\text{App}) \quad + \text{equality rule}$$

$$\frac{\Gamma(x : A) \vdash t : B \qquad \Gamma \vdash s : A}{(\lambda(x : A).\, t)\ s = t[x \mapsto s] : B[x \mapsto s]}\ (\beta) \qquad \frac{\Gamma \vdash f : (x : A) \to B \qquad x \notin FV(f)}{\lambda(x : A).\, f\ x = f : (x : A) \to B}\ (\eta)$$

$$+ \text{ reflexivity, symmetry, and transitivity rules for } =$$

Fig. 2. The core formal rules of UTT, including dependent function types $(x : A) \to B$, an infinite hierarchy of universes $Set_0, Set_1, Set_2, \ldots$, and $\beta\eta$-equality.

Contexts and substitutions. We use Greek capitals $\Gamma, \Delta, \ldots$ for contexts, capitals $T, U, \ldots$ for types, and small letters $t, u, \ldots$ for terms. A list of terms is indicated by a bar above the letter: $\bar{t}$. Contexts double as the type of such a list of terms, so we can write for example $\bar{t} : \Gamma$ where $\Gamma = (m : \texttt{Nat})(p : m \equiv \texttt{zero})$ and $\bar{t} = \texttt{zero refl}$. The simultaneous substitution of the terms $\bar{t}$ for the variables in the context Γ is written as $[\Gamma \mapsto \bar{t}]$. We denote substitutions by small greek letters $\sigma, \tau, \ldots$ The identity substitution is written as $[]$, and the forward composition of two substitutions σ and τ is written as $\sigma; \tau$.

Inductive Families. Inductive families are (dependent) types inductively defined by a number of *constructors*, for example `Nat` is defined by the constructors `zero` : `Nat` and `suc` : `Nat` $\rightarrow$ `Nat`. Inductive families can also have *parameters* and *indices*, for example `Vec` A n is an inductive family with one parameter $A : Set$, one index n : `Nat`, and two constructors `nil` : `Vec` A `zero` and `cons` : (n : `Nat`) $\rightarrow A \rightarrow$ `Vec` A $n \rightarrow$ `Vec` A (`suc` n). A formal treatment of inductive families can be found in [Dyb94]. For our purposes, it suffices to know that inductive families are introduced by the rules given in Fig. 3.

$$\frac{\Gamma\ \textbf{valid}}{\Gamma \vdash D : \Psi\Delta \rightarrow Set_l}\ (\text{Data}) \qquad\qquad \frac{\Gamma\ \textbf{valid}}{\Gamma \vdash c_k : \Psi\Phi_k \rightarrow D\ \bar{\imath}_k}\ (\text{Cons})$$

Fig. 3. Introduction rules for an inductive family D with parameters Ψ, indices Δ, and constructors $c_k : \Phi_k \rightarrow D\ \bar{\imath}_k$ for $k = 1, \ldots, n$.

Definitional and Propositional Equality. In (intensional) type theory, there are two distinct notions of equality. On the one hand, two terms s and t are *definitionally equal* (or *convertible*) if $\Gamma \vdash s = t : T$. On the other hand, two terms s and t are *propositionally equal* if we can prove their equality, i.e. if we can give a term of type $s \equiv_T t$. Propositional equality was introduced by Martin-Löf [ML84]. In UTT, it can be defined as an inductive family with two parameters $A : Set_i$ and $a : A$, one index $b : A$, and one constructor `refl` : $a \equiv_A a$.

When working with type theory in dependently typed languages such as Agda or Coq, it is more convenient to work with definitional equalities rather than propositional ones. This is because (in intensional type theory) definitional equality can be checked automatically, while propositional equality has to be proven and applied manually. When working with terms with free variables however, not all propositionally equal terms are definitionally equal, so the propositional equality is often necessary.

Definitions by Pattern Matching. A definition by pattern matching of a function f consists of a number of equalities called *clauses*, which are of the form $f\ \bar{p} = t$ where $\bar{p}$ is a list of patterns and t is a term called the *right-hand side*. A *pattern* is a term or a list of terms that is built from only (fully applied) constructors and variables, which we call the pattern variables. In dependent pattern matching, patterns can also contain *inaccessible patterns*, which can occur when there is only one type-correct term possible in a given position. As in [Nor07], we mark inaccessible patterns as $\lfloor t \rfloor$. For example, let `Square` n be an inductive family with one index n : `Nat` and one constructor `sq` : (m : `Nat`) $\rightarrow$ `Square` m^2. Then $\lfloor m^2 \rfloor$ (`sq` m) is a pattern of type (n : `Nat`)(p : `Square` n). Any other pattern $\lfloor t \rfloor$ (`sq` m) would be ill-typed, so the use of an inaccessible pattern is justified.

We see patterns as a distinct syntactic class rather than a special kind of terms. We can convert a pattern p to a term by taking the *underlying term* $\lceil p \rceil$

defined as follows:

$$\lceil x \rceil = x \qquad \lceil c\, p_1\, \ldots\, p_n \rceil = c\, \lceil p_1 \rceil\, \ldots\, \lceil p_n \rceil \qquad \lceil \lfloor t \rfloor \rceil = t \tag{6}$$

A term t *matches* a pattern p if there exists a substitution σ such that $\lceil p \rceil \sigma = t$.

A pattern $\bar{p} : \Delta$ is called *linear* if each pattern variable occurs exactly once in an accessible position in $\bar{p}$. It is called *respectful* [GMM06] if for each list of terms $\bar{a} : \Delta$ that matches all the accessible parts of $\bar{p}$, we have that $\bar{a}$ matches all the inaccessible parts of $\bar{p}$ as well. Patterns are required to be linear and respectful in order to have decidable pattern matching in the presence of inaccessible patterns.

Formally, we write $\Gamma | \Phi \vdash \bar{p} : \Delta$ **pattern** to express that, in the context Γ, $\bar{p}$ is a pattern of type Δ with pattern variables from the context Φ. A definition by pattern matching of a function $f : \Delta \to T$ in a context Γ then consists of a set of clauses of the form $f\ \bar{p} = t$ where $\Gamma | \Phi \vdash \bar{p} : \Delta$ **pattern** is linear and respectful and $\Gamma\Phi(f : \Delta \to T) \vdash t : T[\Delta \mapsto \lceil \bar{p} \rceil]$. In order to ensure correctness, definitions by pattern matching are required to have three additional properties:

Completeness For each closed list of terms $\bar{s} : \Delta$, there must be a pattern $\bar{p}$ such that $\bar{s}$ matches $\bar{p}$. This is required in order to have canonicity, i.e. that any closed normal form of an inductive family is constructor-headed.

Termination There can be no $\bar{s} : \Delta$ such that there is an infinite sequence of evaluation steps $f\ \bar{s} \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$ where f occurs in each of the t_i. This is required in order to have strong normalization.

Confluence If $f\ \bar{s} \longrightarrow^* t_1$ and $f\ \bar{s} \longrightarrow^* t_2$, there should exist a term t such that $t_1 \longrightarrow^* t$ and $t_2 \longrightarrow^* t$. This is required in order to have the Church-Rosser property.

If these three requirements are satisfied, we can add f to the theory by the rules given in Fig. 4.

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash f : \Phi \to T}\ (\text{Func}) \qquad \frac{\Gamma \vdash \bar{s} = \lceil \bar{p}_k \rceil \sigma : \Phi}{\Gamma \vdash f\ \bar{s} = t_k\sigma : T[\Phi \mapsto \bar{s}]}\ (\text{Clause})$$

Fig. 4. Rules for a function $f : \Phi \to T$ defined by the clauses $f\ \bar{p}_k = t_k$ for $k = 1, \ldots, n$.

Respectfulness can be checked step by step by context splitting (see Sect. 2.1 of [Nor07]), completeness is checked by constructing a case tree, and termination can be achieved by requiring that definitions are structurally recursive. Confluence is a non-issue as long as first-match semantics are used, because then only one clause is ever applicable at the same time. When we drop the first-match semantics, the checks for respectfulness and termination stay valid, but those for completeness and confluence need to be updated. We will do this in Sect. 5.

Case Trees. Definitions by pattern matching can be represented by a *case tree*. A case tree tells us how the patterns of a definition are built by introducing constructors step by step. Each leaf node of a splitting tree corresponds to a clause of the definition. For example, consider the function `parity` (7) given by Achim Jung on the Agda mailing list[3]. It can be represented by the case tree given in Fig. 5.

$$\underline{m}\; n \begin{cases} \mathtt{zero}\; \underline{n} \begin{cases} \mathtt{zero}\;\mathtt{zero} \mapsto \mathtt{true} \\ \mathtt{zero}\;(\mathtt{suc}\;\underline{n}) \begin{cases} \mathtt{zero}\;(\mathtt{suc}\;\mathtt{zero}) \mapsto \mathtt{false} \\ \mathtt{zero}\;(\mathtt{suc}\;(\mathtt{suc}\;n)) \mapsto \mathtt{parity}\;\mathtt{zero}\;n \end{cases} \end{cases} \\ (\mathtt{suc}\;m)\;\underline{n} \begin{cases} (\mathtt{suc}\;\underline{m})\;\mathtt{zero} \begin{cases} (\mathtt{suc}\;\mathtt{zero})\;\mathtt{zero} \mapsto \mathtt{false} \\ (\mathtt{suc}\;(\mathtt{suc}\;m))\;\mathtt{zero} \mapsto \mathtt{parity}\;m\;\mathtt{zero} \end{cases} \\ (\mathtt{suc}\;m)\;(\mathtt{suc}\;n) \mapsto \mathtt{parity}\;m\;n \end{cases} \end{cases}$$

Fig. 5. This case tree corresponds precisely to the definition of `parity` (7).

$$\begin{array}{llll} \mathtt{parity} : & \mathtt{Nat} \rightarrow & \mathtt{Nat} & \rightarrow \mathtt{Bool} \\ \mathtt{parity} & \mathtt{zero} & \mathtt{zero} & = \mathtt{true} \\ \mathtt{parity} & \mathtt{zero} & (\mathtt{suc}\;\mathtt{zero}) & = \mathtt{false} \\ \mathtt{parity} & \mathtt{zero} & (\mathtt{suc}\;(\mathtt{suc}\;n)) & = \mathtt{parity}\;\mathtt{zero}\;n \\ \mathtt{parity} & (\mathtt{suc}\;\mathtt{zero}) & \mathtt{zero} & = \mathtt{false} \\ \mathtt{parity} & (\mathtt{suc}\;(\mathtt{suc}\;m)) & \mathtt{zero} & = \mathtt{parity}\;m\;\mathtt{zero} \\ \mathtt{parity} & (\mathtt{suc}\;m) & (\mathtt{suc}\;n) & = \mathtt{parity}\;m\;n \end{array} \tag{7}$$

Using case trees has a number of advantages. Firstly, the patterns at the leaves of a case tree always form a covering, hence they are complete. Secondly, they give an efficient method to evaluate functions defined by pattern matching. Thirdly, each internal node in a case tree corresponds exactly to the application of an eliminator for an inductive family, so they are a useful intermediate step in the translation of dependent pattern matching to pure type theory (without pattern matching) as done in [GMM06].

In section 2.2 of [Nor07], it is described how a case tree can be constructed from a given (complete) set of clauses. When dealing with overlapping patterns, the algorithm chooses whatever pattern comes first. In other words, the resulting case tree follows the first-match semantics of pattern matching.

Termination Checking. In order to guarantee termination, functions are required to be *structurally recursive*. This means that the arguments of recursive calls should be *structurally smaller* than the pattern on the left-hand side. The structural order $\prec$ is defined in Fig. 6. For functions with multiple arguments, the function should be structurally recursive on one of its arguments, i.e. there should be some k such that $s_k \prec \lceil p_k \rceil$ for each clause $f\;\bar{p} = t$ and each recursive call $f\;\bar{s}$ in t.

[3] `https://lists.chalmers.se/pipermail/agda/2012/004397.html`, last visited on 15 January 2014.

$$\frac{}{t_i \prec c\, t_1 \,\ldots\, t_n} \qquad\qquad \frac{f \prec t}{f\, s \prec t} \qquad\qquad \frac{r \prec s \qquad s \prec t}{r \prec t}$$

Fig. 6. The structural order $\prec$ can be used to check termination [GMM06]. The most important property of the structural order is that it is well-founded, because this guarantees that structurally recursive functions are indeed terminating.

We lack the space to do justice to the large amount of research on termination checking. For some more sophisticated approaches, see for example size-change termination [LJB01], type-based termination [Bla04], and almost-full relations [VCW12].

3 Problem Statement

When deciding which definitions by pattern matching are allowed, there is a conflict between theory and practice. From a theoretical perspective, we want to be able to write definitions by pattern matching in function of eliminators as in [GMM06], because this guarantees correctness of the definitions. From a practical perspective, we want to be able to write overlapping definitions that follow the first-match semantics, because this reduces the number of clauses required in some cases. In an attempt to reconcile these two goals, [Nor07] allows patterns to overlap but translates definitions to a case tree internally using the first-match semantics. However, the representation of function definitions as case trees and the translation to a case tree specifically introduce a number of new problems which we describe in this section.

Clauses Are Split Too Much. When constructing a case tree from a set of clauses, the constructed case tree is not always the one the user intended. An example of this behavior was given by the definition of `or` (4) in the introduction. As another example, when we translate the definition (7) to a case tree using the algorithm from [Nor07], we won't get the case tree in Fig. 5 but rather the one given in Fig. 7. Note that in this case tree, the constructors are introduced in a different order. The result is that the single clause $\texttt{parity}\ (\texttt{suc}\ m)\ (\texttt{suc}\ n) = \texttt{parity}\ m\ n$ has been split in the following two clauses:

$$\begin{array}{lll}
\texttt{parity}\ (\texttt{suc}\ \texttt{zero}) & (\texttt{suc}\ n) = \texttt{parity}\ \texttt{zero}\ n \\
\texttt{parity}\ (\texttt{suc}\ (\texttt{suc}\ m)) & (\texttt{suc}\ n) = \texttt{parity}\ (\texttt{suc}\ m)\ n
\end{array}$$

This means that a term of the form $\texttt{parity}\ (\texttt{suc}\ m)\ (\texttt{suc}\ n)$, where m and n are free variables, won't evaluate to $\texttt{parity}\ m\ n$, even though it should according to the input clauses. This impedes equational reasoning and can be very confusing to the unsuspecting user. If the last clause was placed first instead, then the correct covering would have been reconstructed. So although the patterns of this definition form a covering, their order nevertheless influences the result!

$$\underline{m}\;n \begin{cases} \texttt{zero}\;\underline{n} \begin{cases} \texttt{zero}\;\texttt{zero} \mapsto \texttt{true} \\ \texttt{zero}\;(\texttt{suc}\;\underline{n}) \begin{cases} \texttt{zero}\;(\texttt{suc}\;\texttt{zero}) \mapsto \texttt{false} \\ \texttt{zero}\;(\texttt{suc}\;(\texttt{suc}\;n)) \mapsto \texttt{parity}\;\texttt{zero}\;n \end{cases} \end{cases} \\ (\texttt{suc}\;\underline{m})\;n \begin{cases} (\texttt{suc}\;\texttt{zero})\;\underline{n} \begin{cases} (\texttt{suc}\;\texttt{zero})\;\texttt{zero} \mapsto \texttt{false} \\ (\texttt{suc}\;\texttt{zero})\;(\texttt{suc}\;n) \mapsto \texttt{parity}\;\texttt{zero}\;n \end{cases} \\ (\texttt{suc}\;(\texttt{suc}\;m))\;\underline{n} \begin{cases} (\texttt{suc}\;(\texttt{suc}\;m))\;\texttt{zero} \mapsto \texttt{parity}\;m\;\texttt{zero} \\ (\texttt{suc}\;(\texttt{suc}\;m))\;(\texttt{suc}\;n) \mapsto \texttt{parity}\;(\texttt{suc}\;m)\;n \end{cases} \end{cases} \end{cases}$$

Fig. 7. In contrast to the case tree in Fig. 5, this case tree of the `parity` function does not include the definitional equality `parity` (`suc` m) (`suc` n) = `parity` m n.

Not All Complete Pattern Sets Form a Covering. The second problem is that not all complete pattern sets form a covering, hence they cannot be represented precisely by a case tree. Consider for example the following definition of `majority` due to Gérard Berry:

$$\begin{array}{llllll} \texttt{majority} : & \texttt{Bool} \to & \texttt{Bool} \to & \texttt{Bool} & \to & \texttt{Bool} \\ \texttt{majority} & \texttt{true} & \texttt{true} & \texttt{true} & = & \texttt{true} \\ \texttt{majority} & x & \texttt{false} & \texttt{true} & = & x \\ \texttt{majority} & \texttt{true} & y & \texttt{false} & = & y \\ \texttt{majority} & \texttt{false} & \texttt{true} & z & = & z \\ \texttt{majority} & \texttt{false} & \texttt{false} & \texttt{false} & = & \texttt{false} \end{array} \tag{8}$$

It is clear that the patterns of this definition are complete and do not overlap, yet there is no case tree representing exactly this definition. Instead, it is translated to the case tree given in Fig. 8. We can see that in the case tree, the clause

$$\underline{x}\;y\;z \begin{cases} \texttt{true}\;\underline{y}\;z \begin{cases} \texttt{true}\;\texttt{true}\;\underline{z} \begin{cases} \texttt{true}\;\texttt{true}\;\texttt{true} \mapsto \texttt{true} \\ \texttt{true}\;\texttt{true}\;\texttt{false} \mapsto \texttt{true} \end{cases} \\ \texttt{true}\;\texttt{false}\;\underline{z} \begin{cases} \texttt{true}\;\texttt{false}\;\texttt{true} \mapsto \texttt{true} \\ \texttt{true}\;\texttt{false}\;\texttt{false} \mapsto \texttt{false} \end{cases} \end{cases} \\ \texttt{false}\;\underline{y}\;z \begin{cases} \texttt{false}\;\texttt{true}\;z \mapsto z \\ \texttt{false}\;\texttt{false}\;\underline{z} \begin{cases} \texttt{false}\;\texttt{false}\;\texttt{true} \mapsto \texttt{false} \\ \texttt{false}\;\texttt{false}\;\texttt{false} \mapsto \texttt{false} \end{cases} \end{cases} \end{cases}$$

Fig. 8. Case tree constructed from the definition of `majority` (8). It does not include the definitional equality `majority` x `false` `true` = x.

`majority` x `false` `true` = x has been split into the following two clauses:

$$\begin{array}{l} \texttt{majority}\;\texttt{true}\;\;\texttt{false}\;\texttt{true} = \texttt{true} \\ \texttt{majority}\;\texttt{false}\;\texttt{false}\;\texttt{true} = \texttt{false} \end{array} \tag{9}$$

So we have lost the definitional equality $\texttt{majority}\ x\ \texttt{false}\ \texttt{true} = x$. Note that no case tree corresponds precisely to the definition (8), so this problem is inherent to the representation of definitions by case trees.

Overlapping Patterns Can Be Useful. It would sometimes be useful to define a function with overlapping clauses that are all interpreted as definitional equalities, for example definition (5) of `plus`. Currently, such definitions are not allowed because the last two clauses are 'unreachable'. Yet in order to evaluate $\texttt{plus}\ m\ \texttt{zero}$ or $\texttt{plus}\ m\ (\texttt{suc}\ n)$ where m is not in constructor form, we need the last two clauses. For example, with definition (5) of `plus`, it is easy to define the function `plus-comm` that proves the commutativity of `plus`:

$$\begin{array}{llll} \texttt{plus-comm} : (m : \texttt{Nat}) \to (n : \texttt{Nat}) \to \texttt{plus}\ m\ n \equiv \texttt{plus}\ n\ m & & & \\ \texttt{plus-comm}\ \ \texttt{zero} & n & = & \texttt{refl} \\ \texttt{plus-comm}\ \ (\texttt{suc}\ m) & n & = & \texttt{cong}\ \texttt{suc}\ (\texttt{plus-comm}\ m\ n) \end{array} \tag{10}$$

In contrast, to give this proof for the standard definition of `plus`, the Agda standard library first needs a lemma to prove that $\texttt{plus}\ m\ (\texttt{suc}\ n) \equiv \texttt{suc}\ (\texttt{plus}\ m\ n)$, and then the proof itself still takes approximately eight lines. So the overlapping patterns of `plus` allow us to give shorter and more straightforward proofs than before. Note that no case tree can contain overlapping patterns, so again this restriction is inherent to the representation by case trees.

4 Allowing More General Pattern Sets

To fix these problems, we extend pattern matching in order to allow more general pattern sets than just coverings. In particular, we allow the patterns in a definition to overlap. Instead of following the first-match semantics, these definitions follow 'any-match semantics', i.e. any clause can be used to evaluate the function at any time. In practice, this means that evaluation of a function application doesn't block when pattern matching gets stuck on a free variable. Instead, evaluation continues with the next clause. This gives us 'what-you-see-is-what-you-get' pattern matching where all clauses hold as definitional equalities.

By extending pattern matching in this way, we solve all three above problems. All clauses are treated as definitional equalities, so their order doesn't matter. We don't need patterns to form a covering, so there is no need to split clauses. Overlapping patterns are allowed, so there is no need to discard them. However, our approach also has some drawbacks:

- First of all, we lose the first-match semantics. This doesn't restrict the functions we can define, but it requires us to write longer definitions in some cases. This problem is unavoidable if we want clauses to be order-independent.
- We also lose the ability to translate definitions to pure type theory with eliminators. To guarantee correctness (completeness, termination, and confluence), we thus need to reason about the definitions directly.

– Finally, we lose the ability to represent functions by case trees, hence the ability to evaluate them efficiently. It is however possible to extend case trees with *catchall subtrees* that allow us to represent these more general definitions by case trees. See the first author's master thesis [Coc13] for a full description.

5 Checking Definitions with Overlapping Patterns

The standard technique for checking termination doesn't depend on the fact that the patterns form a covering, but those for completeness and confluence do. In this section, we describe how to check completeness and confluence in the presence of overlapping patterns.

Completeness. To check whether a set of (overlapping) patterns is complete, we just try to build a case tree for it using the coverage algorithm from section 2.2 of [Nor07]. Because this algorithm can only split patterns or discard them, we know that it preserves completeness. Hence if the construction of a case tree succeeds, we know that the patterns we started from are complete. More formally, we have the following (equivalent) criterion for completeness:

Proposition 1. *Let Δ be a valid context and P be a set of lists of patterns of the same type Δ. If there exists a covering O such that for each $\bar{q} \in O$, there exists a $\bar{p} \in P$ such that $\bar{p} \supseteq \bar{q}$*[4]*, then P is complete.*

Proof. Since the covering O is complete, each closed list of terms $\bar{t} : \Delta$ matches a $\bar{q} \in O$, i.e. there exists a substitution τ such that $\bar{t} = \lceil \bar{q} \rceil \tau$. By assumption, there exists a $\bar{p}$ such that $\bar{p} \supseteq \bar{q}$, i.e. there exists a substitution σ such that $\lceil \bar{p} \rceil \sigma = \lceil \bar{q} \rceil$. Then we have $\bar{t} = \lceil \bar{p} \rceil \sigma\tau$. This holds for any $\bar{t} : \Delta$, hence the set of patterns P is complete. □

The fact that we can reuse the existing coverage algorithm means we don't have to change our intuition about when a function definition is complete. It also means we can re-use existing code for coverage checking.

Confluence. To ensure the confluence of a definition with overlapping patterns, we want that whenever a term matches the patterns of two clauses, then they also give the same result for that term. In order to check whether two patterns overlap, we will use unification. A *unifier* of two terms a and b is a substitution σ such that $a\sigma = b\sigma$. A *most general unifier* of a and b is a unifier σ such that for each other unifier σ', there exists a substitution τ such that $\sigma' = \sigma; \tau$. The question whether unifiers exist is called the *unification problem.* In general, this is an undecidable problem. There exist unification algorithms (see for example

[4] We write $\bar{p} \supseteq \bar{q}$ ($\bar{q}$ is a *specialization* of $\bar{p}$) if there exists a substitution σ on the pattern variables of $\bar{p}$ such that $\lceil \bar{q} \rceil = \lceil \bar{p} \rceil \sigma$.

[McB00]) but they can give up in case the problem is too hard. We say that the algorithm *succeeds positively* if it finds a most general unifier, that it *succeeds negatively* if it concludes there exist no unifiers, and that it *fails* otherwise.

We make the following observation: let $\bar{p}_1$ and $\bar{p}_2$ be two patterns that have a most general unifier σ and let $\bar{p} = \bar{p}_1\sigma = \bar{p}_2\sigma$. Then a term $\bar{t}$ matches $\bar{p}$ if and only if it matches both $\bar{p}_1$ and $\bar{p}_2$. Also, if there is no unifier of $\bar{p}_1$ and $\bar{p}_2$, then there is no term $\bar{t}$ that matches both $\bar{p}_1$ and $\bar{p}_2$. So if we require that the unification of each pair of patterns (with all pattern variables as the flexible variables) succeeds (either positively or negatively) then we are able to check whether two patterns overlap. This is the idea behind the following proposition.

Proposition 2. *Let $f : \Delta \to T$ be defined by a set of clauses which are structurally recursive on the k'th argument. Assume that for each pair of clauses $f\ \bar{p}_1 = t_1$ and $f\ \bar{p}_2 = t_2$ we have that unification of $\bar{p}_1$ and $\bar{p}_2$ succeeds (either positively or negatively). Moreover, assume that if it succeeds positively with result σ, then $t_1\sigma$ and $t_2\sigma$ have the same normal form. Then the definition of f is confluent.*

Proof. Let $\bar{u} : \Delta$ be a normal form, we prove that $f\ \bar{u}$ has a unique normal form by structural induction on the k'th component u_k. So suppose that this is true for all normal forms $\bar{v} : \Delta$ with $v_k \prec u_k$, and suppose $f\ \bar{u} \longrightarrow s_1$ and $f\ \bar{u} \longrightarrow s_2$. Then there exist clauses $f\ \bar{p}_1 = t_1$, $f\ \bar{p}_2 = t_2$ and substitutions τ_1, τ_2 such that $\lceil \bar{p}_1 \rceil \tau_1 = \bar{u} = \lceil \bar{p}_2 \rceil \tau_2$, $t_1\tau_1 = s_1$ and $t_2\tau_2 = s_2$. In particular we have that $\tau = \tau_1; \tau_2$ is a unifier of $\bar{p}_1$ and $\bar{p}_2$, so unification of $\bar{p}_1$ and $\bar{p}_2$ cannot succeed negatively. Unification of $\bar{p}_1$ and $\bar{p}_2$ cannot fail by assumption, hence it must succeed positively with result the most general unifier σ and moreover there must exist a normal form t such that $t_1\sigma \longrightarrow^* t$ and $t_2\sigma \longrightarrow^* t$. Because σ is a most general unifier of $\bar{p}_1$ and $\bar{p}_2$, there exists a substitution τ' such that $\tau = \sigma; \tau'$. This implies $s_1 = t_1\tau = (t_1\sigma)\tau' \longrightarrow^* t\tau'$ and $s_2 = t_2\tau = (t_2\sigma)\tau' \longrightarrow^* t\tau'$. By the induction hypothesis, all recursive calls to f in $t_1\tau$ and $t_2\tau$ have a unique normal form, hence the (shared) normal form $t\tau'$ of $t_1\tau$ and $t_2\tau$ is unique. We can conclude that the definition of f is confluent. $\square$

Note that in order to check confluence of a recursive function, we need to know that the definition has already passed the termination checker. This is because we need to evaluate the function in question in order to check confluence.

It can happen that the unification of two patterns fails while checking confluence. However, unification of patterns consisting of only constructors and variables always succeeds (either positively or negatively). So this problem can only occur if an inaccessible pattern overlaps with a constructor pattern or another inaccessible pattern.

6 Implementation and Examples

Our extended form of pattern matching, as well as the confluence checker, have been implemented as an experimental modification to the Agda compiler. The implementation allows choosing between the standard semantics and ours for each definition separately by the use of a new keyword `overlapping`. We do not give the details of the implementation here, but instead give some examples of definitions with overlapping patterns. In particular, we add extra evaluation rules to some standard definitions. This can make it easier to prove propositions that mention these functions, as in the proof of `plus-comm` (10). We also give an example where our confluence check fails unexpectedly.

Concatenation of Vectors. Here is a definition of the concatenation `concat` on vectors that uses overlapping patterns:

$$\begin{array}{llllll}
\texttt{concat} : (m : \texttt{Nat})\ (n : \texttt{Nat})\ (v : \texttt{Vec}\ A\ m)\ (w : \texttt{Vec}\ A\ n) \rightarrow \texttt{Vec}\ A\ (\texttt{plus}\ m\ n) \\
\texttt{concat}\ \lfloor\texttt{zero}\rfloor & n & \texttt{nil} & w & = w \\
\texttt{concat}\ m & \lfloor\texttt{zero}\rfloor & v & \texttt{nil} & = v \\
\texttt{concat}\ \lfloor\texttt{suc}\ m\rfloor & n & (\texttt{cons}\ m\ a\ v) & w & = \texttt{cons}\ m\ a\ (\texttt{concat}\ m\ n\ v\ w)
\end{array} \tag{11}$$

Note that for the first clause to be of correct type, we need that $\texttt{plus}\ \texttt{zero}\ n = n$; while for the second clause we need that $\texttt{plus}\ m\ \texttt{zero} = m$. So this definition of `concat` relies upon the fact that the definition of `plus` has overlapping clauses.

Transitivity of the Propositional Equality. The definition of the propositional equality $\equiv_A$ as an inductive family only provides reflexivity of the relation. In order to prove that $\equiv_A$ is symmetric and transitive, we have to give a proof ourselves. For example, here is a proof of transitivity:

$$\begin{array}{lllllll}
\texttt{trans} : (x : A) & (y : A) & (z : A) & (p : x \equiv y) & (q : y \equiv z) & \rightarrow & x \equiv z \\
\texttt{trans}\ \ \lfloor y\rfloor & \lfloor y\rfloor & z & \texttt{refl} & q & = & q \\
\texttt{trans}\ \ x & \lfloor y\rfloor & \lfloor y\rfloor & p & \texttt{refl} & = & p
\end{array} \tag{12}$$

We again use overlapping patterns in order to increase the number of evaluation rules. This ensures that both proofs of the form $\texttt{trans}\ \texttt{refl}\ p$ and $\texttt{trans}\ p\ \texttt{refl}$ are automatically simplified to p, saving us from proving them ourselves. This also shows that the confluence checker works in the presence of inaccessible patterns.

A Counterexample: Multiplication. Here is another function on natural numbers, multiplication:

$$\begin{array}{llll}
\texttt{mult} : \texttt{Nat} \rightarrow & \texttt{Nat} & \rightarrow & \texttt{Nat} \\
\texttt{mult}\ \ \texttt{zero} & y & = & \texttt{zero} \\
\texttt{mult}\ \ (\texttt{suc}\ x) & y & = & \texttt{plus}\ (\texttt{mult}\ x\ y)\ y \\
\texttt{mult}\ \ x & \texttt{zero} & = & \texttt{zero} \\
\texttt{mult}\ \ x & (\texttt{suc}\ y) & = & \texttt{plus}\ x\ (\texttt{mult}\ x\ y)
\end{array} \tag{13}$$

Let us focus on the confluence of the second and the fourth clause. After unification of the patterns, the right-hand sides become respectively:

$$\texttt{plus}\ (\texttt{mult}\ x\ (\texttt{suc}\ y))\ (\texttt{suc}\ y) \longrightarrow^* \texttt{suc}\ (\texttt{plus}\ (\texttt{plus}\ x\ (\texttt{mult}\ x\ y))\ y)$$

$$\texttt{plus}\ (\texttt{suc}\ x)\ (\texttt{mult}\ (\texttt{suc}\ x)\ y) \longrightarrow^* \texttt{suc}\ (\texttt{plus}\ x\ (\texttt{plus}\ (\texttt{mult}\ x\ y)\ y))$$

We see that the right-hand sides do not have the same normal form, but are only equal *up to associativity* of `plus`. Hence this definition does not satisfy our criterion for confluence (Proposition 2). It is however possible to prove that the right-hand sides are *propositionally* equal. But to obtain confluence, we need them to have the same normal form, i.e. they must be *definitionally* equal. To solve this problem, we would have to introduce a new evaluation rule of the form

$$\texttt{plus}\ (\texttt{plus}\ x\ y)\ z \longrightarrow \texttt{plus}\ x\ (\texttt{plus}\ y\ z)$$

Such rules are currently not allowed in type theory, and it is not clear how to add them in a sound way. Hence we refrain from allowing definitions such as (13) in the current work.

7 Link with Non-overlapping Definitions

We have shown that overlapping function definitions can be useful, but we also have to worry about soundness. For definitions by pattern matching whose patterns form a covering, this is done by translating the definition to repeated application of eliminators [GMM06]. If the patterns of a definition do not form a covering however, there is no hope to proceed in this way.

In this section, we prove that each new function definition we introduce is equivalent to an old one. In order to formulate the proposition, we first have to define what we mean by 'equivalent'. It is not realistic to ask that they are definitionally or propositionally equal, because both are *intensional* equalities: they care about how functions are defined, not just about their values. To solve this problem, we assume the functional extensionality axiom, which expresses that two functions are equal when they have equal values for equal inputs. This is achieved by adding for each pair of functions $f_1, f_2 : (x : A) \to B\ x$ the following constant:

$$\texttt{Ext} : ((x : A) \to f_1\ x \equiv f_2\ x) \to f_1 \equiv f_2 \qquad (14)$$

This constant was introduced by [Hof95]. Now we can state our main theorem:

Theorem 3. *Assume the functional extensionality axiom (14). If a function $\Gamma \vdash f : \Delta \to T$ is defined by a set of clauses that satisfy the criteria for completeness (see Proposition 1), termination (i.e. the definition is structurally recursive), and confluence (see Proposition 2); then we can define a function $\Gamma \vdash f' : \Delta \to T$ whose patterns form a covering such that $\Gamma \vdash eq_f : f \equiv f'$ where eq_f only contains functions whose patterns form a covering as well.*

The equality proof eq_f given by this theorem is internal to the language, rather than meta-theoretical. In principle, this could cause problems because we don't prove consistency of the extended language. However, note the following:

- Functions with overlapping patterns cannot occur inside the equality proof. So possible inconsistencies arising from non-confluent definitions do not invalidate the theorem.

– The function f is not required to be terminating, but only structurally recursive, which is easily checked and requires no further proof. It would be better to be independent of the specific termination criterion, but this would introduce a circularity in the proof.
– While we need reductions in order to typecheck a function and hence to check its completeness, a function can never occur in its own type. Hence we do not need to know the definition is confluent in order to check completeness.

In order to prove this theorem, we use the heterogeneous equality $a \cong_{A,B} b$ introduced by McBride [McB00]. It allows the expression of equality between terms of different types, but still only allows a proof if the types are equal. Heterogeneous equality can be defined as an inductive family with two parameters $A : Set_i$ and $a : A$, *two* indices $B : Set_i$ and $b : B$, and one constructor `refl` $: a \cong_{A,A} a$. In contrast to [McB00], this definition uses the standard elimination principle (which McBride calls `eqIndElim`). We will work with the heterogeneous equality by means of pattern matching, this is equivalent with using `eqIndElim` together with the K axiom [GMM06]. We will use the following fact about the heterogeneous equality:

– For any type A and terms $x, y : A$, we have:

$$\texttt{hom-to-het} : x \equiv y \rightarrow x \cong y \tag{15}$$
$$\texttt{het-to-hom} : x \cong y \rightarrow x \equiv y \tag{16}$$

Assuming extensionality, we additionally have the following facts:

– For all $f_1 : (x : A_1) \rightarrow B_1\ x$ and $f_2 : (x : A_2) \rightarrow B_2\ x$, we have:

$$\begin{aligned}\lambda\texttt{-cong} : (A_1 \cong A_2) \rightarrow \\ ((x_1 : A_1)(x_2 : A_2) \rightarrow x_1 \cong x_2 \rightarrow f_1\ x_1 \cong f_2\ x_2) \rightarrow \\ f_1 \cong f_2\end{aligned} \tag{17}$$

– For all $t_1 : A_1$, $t_2 : A_2$, $f_1 : (x : A_1) \rightarrow B_1$ and $f_2 : (x : A_2) \rightarrow B_2$, we have:

$$\begin{aligned}\texttt{ap-cong} : ((x_1 : A_1)(x_2 : A_2) \rightarrow x_1 \cong x_2 \rightarrow B_1\ x_1 \cong B_2\ x_2) \rightarrow \\ f_1 \cong f_2 \rightarrow t_1 \cong t_2 \rightarrow f_1\ t_1 \cong f_2\ t_2\end{aligned} \tag{18}$$

– For all $B_1 : A_1 \rightarrow Set_i$ and $B_2 : A_1 \rightarrow Set_i$ we have:

$$\begin{aligned}\Pi\texttt{-cong} : (A_1 \cong A_2) \rightarrow \\ ((x_1 : A_1)(x_2 : A_2) \rightarrow x_1 \cong x_2 \rightarrow B_1\ x_1 \cong B_2\ x_2) \rightarrow \\ ((x_1 : A_1) \rightarrow B_1\ x_1) \cong ((x_2 : A_2) \rightarrow B_2\ x_2)\end{aligned} \tag{19}$$

The last three facts are used mainly as a tool to 'push' our (heterogeneous) propositional equalities through all syntactic constructs. For a machine-checked proof of these facts in Agda, please refer to the appendix.

Proof (of Theorem 3). We start by giving the definition of the function f'. Let P be the set of patterns in the definition of f. Because the clauses of f satisfy the criterion for completeness (Proposition 1), there exists a covering O such that for each $\bar{q} \in O$, there exists a $\bar{p} \in P$ such that $\bar{p} \supseteq \bar{q}$. In other words, for all $\bar{q} \in O$ there exists a clause $f\ \bar{p} = t$ of f and a substitution σ such that $\bar{p}\sigma = \bar{q}$. This means we have $\Gamma\Psi(f : \Delta \to T) \vdash t : T[\Delta \mapsto \lceil \bar{p} \rceil]$ where Ψ is the context of pattern variables of $\bar{p}$. Let t' be the term t where all occurrences of f have been replaced by f'. The function f' is defined by the clauses $f'\ \bar{q} = t'\sigma$ for all $\bar{q} \in O$. We check that this is a valid definition:

- Let Φ be the context of pattern variables of $\bar{q}$. We have $\Gamma\Phi(f' : \Delta \to T) \vdash t'\sigma : T[\Delta \mapsto \lceil \bar{q} \rceil]$ by α-renaming and the fact that $\bar{p}\sigma = \bar{q}$, so the clauses of f' are valid.
- The set of patterns O is a covering, hence the patterns are complete.
- The arguments of all recursive calls $f\ \bar{s}$ in the right-hand side of a clause $f\ \bar{p} = t$ satisfy $\bar{s} \prec \lceil \bar{p} \rceil$. Note that if $s \prec t$, then also $s\sigma \prec t\sigma$ for any substitution σ (by induction on the definition of $\prec$). This gives us that $\bar{s}\sigma \prec \lceil \bar{p} \rceil\sigma = \lceil \bar{q} \rceil$. This implies that the definition of f' is structurally recursive, hence it is terminating.
- The patterns in O do not overlap, hence the definition of f' is confluent.

Now we define $\tilde{eq}_f$ such that $\Gamma \vdash \tilde{eq}_f : f \cong f'$. By extensionality (14) it is sufficient to give a term $\Gamma \vdash \tilde{eq}_{f(\Delta)} : \Delta \to f\ \Delta \cong f'\ \Delta$. In order to do this, we use pattern matching with the same pattern set O used in the definition of f'. Let $\Gamma|\Phi \vdash \bar{q} : \Delta$ **pattern** be one of these patterns, the return type of $\tilde{eq}_{f(\Delta)}$ for that pattern becomes $f\ \lceil \bar{q} \rceil \cong f'\ \lceil \bar{q} \rceil$.

On the one hand, by definition of O there exists a clause $f\ \bar{p} = t$ of f and a substitution σ such that $\bar{p}\sigma = \bar{q}$. This implies that $\Gamma \vdash f\ \lceil \bar{q} \rceil = t\sigma : T[\Delta \mapsto \lceil \bar{q} \rceil]$. On the other hand, there is a clause $f'\ \bar{q} = t'\sigma$ of f', hence $\Gamma \vdash f'\ \lceil \bar{q} \rceil = t'\sigma : T[\Delta \mapsto \lceil \bar{q} \rceil]$. So we are left to give a term of type $t\sigma \cong t'\sigma$ in the context $\tilde{\Gamma} = \Gamma\Phi(\tilde{eq}_{f(\Delta)} : \Delta \to f\ \Delta \cong f'\ \Delta)$.

Note that the bound variables in t and t' with the same name do not necessarily have the same type, because occurrences of f in the types have been replaced by f'. In order to avoid confusion between these variables, we α-rename all bound variables x in t' to their primed variants x'.

In order to proceed, we first fix some notations. Let Ξ be a context such that $\tilde{\Gamma}\Xi$ **valid**. We denote with Ξ' the context Ξ where each variable x has been replaced by its primed version x' and each occurrence of f has been replaced by f'. If $\tilde{\Gamma}\Xi \vdash a : A$, then a' denotes the term a where each variable from the context Ξ has been replaced by x' and each occurrence of f has been replaced by f'. Note that $\tilde{\Gamma}\Xi' \vdash a' : A'$, and that this can be proven by using the same tree of inference rules. One further notation we use is $\Xi \cong \Xi'$ for the context expressing pairwise equality between the variables in Ξ and Ξ'. For example, if $\Xi = (n : \texttt{Nat})(v : \texttt{Vec}\ n)$ and $\Xi' = (n' : \texttt{Nat})(v' : \texttt{Vec}\ n')$, then $\Xi \cong \Xi' = (eq_n : n \cong n')(eq_v : v \cong v')$.

In order to prove $t\sigma \cong t'\sigma$ in the context $\tilde{\Gamma}$, we give for all contexts Ξ and all terms $\tilde{\Gamma}\Xi \vdash a : A$ a proof $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_a : a \cong a'$. As long as a is not a recursive call of the form $f\ \bar{u}$, we proceed by induction on the derivation of

$\tilde{\Gamma}\Xi \vdash a : A$ (and hence also that of $\tilde{\Gamma}\Xi' \vdash a' : A'$). See Fig. 2, Fig. 3, and Fig. 4 for the relevant rules.

Var rule In this case we have $a = x$ for some variable x from the context $\tilde{\Gamma}\Xi$. If it comes from $\tilde{\Gamma}$, we have $a' = x$ and hence $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \texttt{refl} : x \cong x$. If on the other hand it comes from Ξ, we have $a' = x'$ and $eq_x : x \cong x' \in (\Xi \cong \Xi')$, hence $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_x : x \cong x'$ (where $eq_x : x \cong x' \in \Xi \cong \Xi'$).

=Ty rule In this case we just proceed with the induction on the derivation of the first assumption of the rule.

Set rule In this case we have $a = Set_i$ for some i, hence also $a' = Set_i$. So we have $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \texttt{refl} : Set_i \cong Set_i$.

Π **rule** In this case we have $a = (x : U) \to V$ and $a' = (x : U') \to V' = (x' : U') \to V'[x \mapsto x']$. By the induction hypothesis, we have $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_U : U \cong U'$ and $\tilde{\Gamma}\Xi(x : U)\Xi'(x' : U')(\Xi \cong \Xi')(eq_x : x \cong x') \vdash eq_V : V \cong V'[x \mapsto x']$. This gives us $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \Pi\texttt{-cong}\ eq_U\ (\lambda x\ x'\ eq_x.\ eq_V) : (x : U) \to V \cong (x' : U') \to V'[x \mapsto x']$.

λ **rule** In this case we have $a = \lambda(x : U).\ v$ and $a' = \lambda(x : U').\ v' = \lambda(x' : U').\ v'[x \mapsto x']$. By the induction hypothesis, we have $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_U : U \cong U'$ and $\tilde{\Gamma}\Xi(x : U)\Xi'(x' : U')(\Xi \cong \Xi')(eq_x : x \cong x') \vdash eq_v : v \cong v'[x \mapsto x']$. This gives us $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \lambda\texttt{-cong}\ eq_U\ (\lambda x\ x'\ eq_x.\ eq_v) : \lambda(x : U).\ v \cong \lambda(x' : U').\ v'[x \mapsto x']$.

App rule In this case we have $a = g\ u$ and $a' = g'\ u'$. By the induction hypothesis, we have $\tilde{\Gamma}\Xi(x : U)\Xi'(x' : U')(\Xi \cong \Xi')(eq_x : x \cong x') \vdash eq_V : V \cong V'[x \mapsto x']$, $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_g : g \cong g'$, and $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_u : u \cong u'$. This gives us $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \texttt{ap-cong}\ (\lambda x\ x'\ eq_x.\ eq_V)\ eq_g\ eq_u : g\ u \cong g'\ u'$.

Cons rule In this case we have $a = c$ and $a' = c$ for a constructor c. This gives us $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \texttt{refl} : c \cong c$.

Data rule In this case we have $a = D$ and $a' = D$ for an inductive family D. Hence we have $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \texttt{refl} : D \cong D$.

Func rule In this case we have $a = g$ and $a' = g$ for a defined function g distinct from f and f'. Then we have $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \texttt{refl} : g \cong g$.

In the end, we reach a recursive call: $a = f\ \bar{u}$ and $a' = f'\ \bar{u}'$. In this case, we recursively call the proof $\tilde{eq}_{f(\Delta)}$ which we are in the process of defining: $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash \tilde{eq}_{f(\Delta)}\ \bar{u} : f\ \bar{u} \cong f'\ \bar{u}$. This call is structurally recursive because the recursive call to f in a is. By continuing the induction as above we also get $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_{\bar{u}} : \bar{u} \cong \bar{u}'$. By applying `ap-cong` repeatedly, we get $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_{f'(\bar{u})} : f'\ \bar{u} \cong f'\ \bar{u}'$, and by transivity of $\cong$, we get $\tilde{\Gamma}\Xi\Xi'(\Xi \cong \Xi') \vdash eq_{f(\bar{u})} : f\ \bar{u} \cong f'\ \bar{u}'$, completing the definition of $\tilde{eq}_{f(\Delta)}$ and hence also that of $\tilde{eq}_f$. Finally, by `het-to-hom` (16), we get eq_f such that $\Gamma \vdash eq_f : f \equiv f'$, finishing the proof. □

8 Related Work

Dependent pattern matching was introduced by Coquand in [Coq92]. A big step toward its practical usefulness was the introduction of the 'with' construct by [MM04]. On a more fundamental level, [GMM06] shows that definitions by dependent pattern matching can be translated to pure type theory with the K axiom. Real languages with dependent pattern matching include Agda [Nor07], Coq [Soz10], and Idris [Bra13].

- In [Ken90], *tightest-match semantics* for overlapping patterns are used. To ensure confluence, they require for each pair of overlapping patterns that their unification is also part of the definition. In contrast to our current work, they do not look at the right-hand sides to check confluence.
- In the Calculus of Algebraic Constructions [BJO99] general well-typed rewriting rules are allowed. However, in order to prove confluence they have to assume that the left-hand sides of the rewrite rules do not overlap.
- In deduction modulo [DHK03], overlapping rewriting rules are allowed, but confluence is usually assumed or proven manually.
- In systems based on the LF logical framework and the $\lambda\Pi$-calculus (for example Twelf [PS99]), there can be overlapping clauses, but definitions are not required to be confluent. Instead backtracking is utilized to generate all possible solutions.
- In Isabelle/HOL, it is possible to define functions by pattern matching such that the result doesn't depend on the order of the patterns [Kra06]. In contrast to our work, they don't deal with dependent pattern matching, and they don't give a concrete algorithm for confluence checking.
- Even though we provide more definitional equalities than the standard formulation of pattern matching, some will always be missing. Another possibility would be to add a better support for coercion by propositional equality proofs, as supported for example by OTT [AMS07].
- The recent work on adding equations for neutral terms [ABM13] starts from a motivation similar to ours, but doesn't focus on pattern matching in specific.

9 Conclusion and future work

The main goal of this paper is to make dependent pattern matching more intuitively usable for specialists and non-specialists alike. We do this by extending the semantics of pattern matching in order to allow overlapping patterns. Because all clauses are interpreted as definitional equalities, these definitions behave as one would expect them to. This also makes pattern matching more amenable to equational reasoning. Type theory supports equational reasoning in the language itself by means of the identity type, so this is not just a theoretical advantage, but also a practical one.

In practice, a typical user would probably start by giving a non-overlapping definition and add overlapping clauses when he has a need for them. For example, when giving the clause `concat` $v\ \epsilon = v$ for the concatenation operator on vectors, the type checker complains that the length `plus` n `zero` of the left-hand side does not equal the length n of the right-hand side. The user can then add the clause `plus` n `zero` $= n$ to the definition of `plus`, after which the clause for `concat` passes the type checker. This blends well with the typical interactive development of dependently typed programs in dependently typed programming languages.

The current implementation is still very experimental. It would be interesting to give a full implementation that is compatible with extensions of pattern matching such as wildcard patterns, 'with'-expressions [MM04], and coinductive data types. It should also be possible to implement the pattern matching described in this paper in other languages with dependent pattern matching such as Coq.

One limit to our approach is that the confluence checker doesn't always see that a definition is confluent. This occurs when inaccessible patterns overlap with constructor patterns or other inaccessible patterns. This could be solved by improving the unification algorithm for patterns. Another case where the confluence check fails, is the definition of multiplication (13). This problem is not easily solved by improving the confluence checker, however. Rather, it depends crucially on the question whether we want to see $l + (m + n)$ and $(l + m) + n$ as 'the same' even if l, m and n are free variables.

When designing a dependently-typed programming language, a balance needs to be found w.r.t. the definitional equality. It typically includes at least β-equivalence for functions, but e.g. Agda additionally has definitional η-equivalence for functions and record types [Nor07]. Strengthening definitional equality generally increases programmer convenience but makes equality and type-checking harder for the compiler to decide and may exclude certain models of the theory. When adding functions defined by pattern matching to the theory, definitional equality needs to be extended with their computational behaviour as in the Clause rule of Fig. 4. In this setting, our work can be seen as allowing functions with overlapping reduction rules that cannot be reduced to the non-overlapping rules of data type eliminators. Our new compromise is that we allow overlapping reduction rules as long as confluence can be checked definitionally. We think our approach strikes an interesting new balance between having too little and too many definitional equalities: have any less evaluation rules, and overlapping clauses cannot all hold as definitional equalities; have any more, and extra equalities have to be introduced to regain confluence.

As with any modification to type theory, there is the question of soundness. We think that Theorem 3 gives a step in the right direction, but it is an interesting question whether any extra requirements are needed in order to give a definitive answer. A practical use of this theorem is *program extraction*: since we have $f \cong f'$, these functions both give the same results for *closed* arguments. In a compiled program, only closed terms are evaluated so we can freely replace f by f'. Because f' can be compiled to a case tree, this increases the efficiency of the extracted program.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, and by the Research Foundation - Flanders under grant number G004321N. Jesper Cockx and Dominique Devriese both hold a Ph.D. fellowship of the Research Foundation - Flanders (FWO).

References

Ab12. A. Abel, *Agda: equality.* (`http://www2.tcs.ifi.lmu.de/~abel/Equality.pdf`).

ABM13. G. Allais, P. Boutillier, and C. McBride, *New equations for neutral terms.* Dependently-Typed Programming, 2013.

AMS07. T. Altenkirch, C. McBride, and W. Swierstra, *Observational equality, now!* Programming languages meets program verification, 2007.

BJO99. F. Blanqui, J. Jouannaud, and M. Okada, *The calculus of algebraic constructions.* Rewriting Techniques and Applications, 1999.

Bla04. F. Blanqui, *A type-based termination criterion for dependently-typed higher-order rewrite systems.* Rewriting Techniques and Applications, 2004.

BP85. L. Bachmair, and D. A. Plaisted, *Termination orderings for associative-commutative rewriting systems.* Journal of Symbolic Computation 1.4, 1985.

Bra13. E. Brady, *Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation.* JFP 23.5, 2013.

Coc13. J. Cockx, *Overlapping and order-independent patterns in type theory.* Master thesis, KU Leuven, 2013.

Coq92. T. Coquand, *Pattern matching with dependent types.* Types for proofs and programs, 1992.

DHK03. G. Dowek, T. Hardin, and C. Kirchner, *Theorem proving modulo.* Journal of Automated Reasoning, 2003.

Dyb94. P. Dybjer, *Inductive families.* Formal Aspects of Computing 6.4, 1994.

GMM06. H. Goguen, C. McBride, and J. McKinna, *Eliminating dependent pattern matching.* Algebra, Meaning, and Computation, 2006.

Hof95. M. Hofmann, *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh, 1995.

Hud89. P. Hudak, *Conception, evolution, and application of functional programming languages.* ACM Computing Surveys 21.3, 1989.

Ken90. R. Kennaway, *The specificity rule for lazy pattern-matching in ambiguous term rewrite systems.* ESOP '90 (LNCS 432), 1990.

Kra06. A. Krauss, *Partial recursive functions in higher-order logic.* Automated Reasoning (2006).

LJB01. C. S. Lee, N. D. Jones, and A. M. Ben-Amram, *The size-change principle for program termination.* ACM SIGPLAN Notices 36.3, 2001.

Luo94. Z. Luo, *Computation and reasoning: a type theory for computer science.* International Series of Monographs on Computer Science 11, 1994.

McB00. C. McBride, *Dependently typed functional programs and their proofs.* PhD thesis, University of Edinburgh, 2000.

ML84. P. Martin-Löf, *Intuitionistic type theory.* Studies in Proof Theory 1, 1984.

MM04. C. McBride, and J. McKinna, *The view from the left.* JFP 14.1, 2004.

Nor07. U. Norell, *Towards a practical programming language based on dependent type theory.* PhD Thesis, Chalmers University of Technology, 2007.

PS99. F. Pfenning, and C. Schürmann, *System description: Twelf - a meta-logical framework for deductive systems.* CADE 16, 1999.

Soz10. M. Sozeau, *Equations: A dependent pattern-matching compiler.* ITP, 2010.

VCW12. D. Vytiniotis, T. Coquand, and D. Wahlstedt, *Stop when you are almost-full.* ITP (LNCS 7406), 2012.

Appendix: Agda Code on Heterogeneous Equality and Functional Extensionality

The following is a construction in Agda of `hom-to-het` (15), `het-to-hom` (16), λ-`cong` (17), `ap-cong` (18), and Π-`cong` (19).

```
open import Level using ()

data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x

data _≅_ {a} {A : Set a} (x : A) : {A’ : Set a} → A’ → Set a where
  refl : x ≅ x

hom-to-het : ∀ {a} {A : Set a} {x y : A} → x ≡ y → x ≅ y
hom-to-het refl = refl

het-to-hom : ∀ {a} {A : Set a} {x y : A} → x ≅ y → x ≡ y
het-to-hom refl = refl

equal-type : ∀ {a} {A : Set a} {x : A}  {A’ : Set a} {x’ : A’} →
               x ≅ x’ → A ≡ A’
equal-type refl = refl

equal-types : ∀ {a b} {A : Set a} {B₁ B₂ : A → Set b} →
                {f₁ : (x : A) → B₁ x} {f₂ : (x : A) → B₂ x} →
                ((x : A) → f₁ x ≅ f₂ x) → (x : A) →
                 B₁ x ≡ B₂ x
equal-types eqfx x = equal-type (eqfx x)

ap : ∀ {a b} {A : Set a} {t₁ t₂ : A} →
       {B₁ : A → Set b} → {B₂ : A → Set b} →
       {f₁ : (x : A) → B₁ x} → {f₂ : (x : A) → B₂ x} →
       B₁ ≅ B₂ → f₁ ≅ f₂ → t₁ ≅ t₂ → f₁ t₁ ≅ f₂ t₂
ap eqB eqf refl with eqB  | eqf
ap eqB eqf refl    | refl | refl = refl

postulate Ext : ∀ {a b} {A : Set a} {B : A → Set b}
                  {f₁ f₂ : (x : A) → B x} →
                  ((x : A) → f₁ x ≡ f₂ x) → f₁ ≡ f₂

Ext’: ∀ {a b} {A : Set a} {B₁ B₂ : A → Set b}
        {f₁ : (x : A) → B₁ x} {f₂ : (x : A) → B₂ x} →
        ((x : A) → f₁ x ≅ f₂ x) → f₁ ≅ f₂
Ext’ eqfx with Ext (equal-types eqfx)
Ext’ eqfx | refl = hom-to-het (Ext (λ x → het-to-hom (eqfx x)))
```

```
λ-cong : ∀ {a b} {A₁ A₂ : Set a}
           {B₁ : A₁ → Set b} {B₂ : A₂ → Set b}
           {f₁ : (x : A₁) → B₁ x} {f₂ : (x : A₂) → B₂ x} →
           (A₁ ≅ A₂) →
           ((x₁ : A₁)(x₂ : A₂) → x₁ ≅ x₂ → f₁ x₁ ≅ f₂ x₂) →
            f₁ ≅ f₂
λ-cong refl eqfx = Ext' (λ x → eqfx x x refl)

ap-cong : ∀ {a b} {A₁ A₂ : Set a} {t₁ : A₁} {t₂ : A₂} →
            {B₁ : A₁ → Set b} → {B₂ : A₂ → Set b} →
            {f₁ : (x : A₁) → B₁ x} {f₂ : (x : A₂) → B₂ x} →
            ((x₁ : A₁)(x₂ : A₂) → x₁ ≅ x₂ → B₁ x₁ ≅ B₂ x₂) →
             f₁ ≅ f₂ → t₁ ≅ t₂ → f₁ t₁ ≅ f₂ t₂
ap-cong eqBx eqf refl = ap (Ext' (λ x → eqBx x x refl)) eqf refl

Π-cong : ∀ {a b} {A₁ A₂ : Set a}
            {B₁ : A₁ → Set b} {B₂ : A₂ → Set b} →
            (A₁ ≅ A₂) →
            ((x₁ : A₁)(x₂ : A₂) → x₁ ≅ x₂ → B₁ x₁ ≅ B₂ x₂) →
            ((x : A₁) → B₁ x) ≅ ((x : A₂) → B₂ x)
Π-cong refl eqBx = ap refl refl (Ext' (λ x → eqBx x x refl))
```