KU LEUVEN

FACULTY OF SCIENCE

Overlapping and order-independent patterns in type theory

Jesper COCKX

Supervisor: *Prof. Dr. Ir. Frank Piessens*
Mentor: *Dominique Devriese*

Thesis presented in
fulfillment of the requirements
for the degree of Master of Science
in Mathematics

Academic year 2012-2013

Abstract

Pattern matching is a mechanism to write programs by case distinction and recursion in functional programming languages. In a language based on type theory with dependent types, pattern matching allows us to write not just programs, but also proofs. In order to check whether these proofs are correct, certain restrictions are put on definitions by pattern matching [Coq92]. These restrictions allow us to write definitions by pattern matching in function of the theoretically simpler eliminators [GMM06]. This ensures that definitions by pattern matching are correct, but also limits the expressiveness of the language.

One of these restrictions is that patterns must form a covering, i.e. are created by repeatedly splitting on a pattern variable. Some dependently typed languages with pattern matching like 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. However, the result of the translation depends on the order of the clauses, even when the patterns do not overlap. This can lead to unexpected results for a user who doesn't know the internal workings of this translation. This is a sign of bad abstraction.

This thesis is concerned with making dependent pattern matching more intuitive for the user. We do this by interpreting each clause directly as a definitional equality, even when the patterns overlap. In particular, our interpretation doesn't depend on the order of the patterns. This allows us to give definitions with overlapping patterns, which can be used to extend a function with extra evaluation rules. To interpret pattern matching in this way, we lift the restriction that the patterns must form a covering. Instead of this restriction, we give a more general criterion for completeness. In order to ensure correctness in the presence of overlapping patterns, we also give a criterion for confluence.

By making all clauses hold as definitional equalities, definitions by pattern matching feel more like mathematical definitions, rather than program instructions. However, we lose the ability to translate pattern matching to the use of eliminators, making it more complex to understand theoretically. In order to reduce this loss, we will prove a theoretical result that gives an equivalence with non-overlapping definitions.

Abstract (Dutch)

Pattern matching is een mechanisme om functionele programma's te schrijven aan de hand van gevalsonderscheid en recursie. In een taal gebaseerd op typetheorie met dependent types kunnen we met pattern matching niet alleen programma's schrijven, maar ook bewijzen. Om de correctheid van deze bewijzen te garanderen, worden er bepaalde beperkingen opgelegd aan definities met pattern matching [Coq92]. Deze beperkingen zorgen ervoor dat definities met pattern matching kunnen geschreven worden in functie van de theoretisch gezien eenvoudigere eliminatoren [GMM06]. Dit verzekert ons dat definities met pattern matching correct zijn, maar beperkt tegelijk de uitdrukkingskracht van de taal.

Eén van deze beperkingen is dat de patterns een overdekking moeten vormen, i.e. opgebouwd door herhaaldelijk patterns te splitsen op een patternvariabele. Sommige talen met dependent types en pattern matching zoals Agda [Nor07] laten algemenere patternverzamelingen toe, maar vertalen deze intern naar een overdekking. Bij deze vertaling worden overlappende patterns behandeld op basis van het first-matchprincipe. Het resultaat van deze vertaling hangt af van de volgorde van de clauses, zelfs indien de patterns niet overlappen. Deze vertaling kan dus leiden tot onverwachte resultaten voor een gebruiker die de interne werking ervan niet kent. Dit is een teken van slechte abstractie.

Het doel van deze thesis is om pattern matching intuïtiever te maken voor de gebruiker. Dit doen we door elke clause te interpreteren als een definitionele gelijkheid, zelfs indien de patterns overlappen. In het bijzonder hangt deze interpretatie dus niet af van de volgorde van de clauses. Deze interpretatie laat ons ook toe om definities met overlappende patterns te geven, wat nuttig is om definities uit te breiden met extra evaluatieregels. Om pattern matching op deze manier te kunnen interpreteren, heffen we de beperking op dat de patterns een overdekking moeten vormen. In plaats van deze beperking geven we een algemener criterium voor volledigheid. Om correctheid te garanderen in de aanwezigheid van overlappende patterns, geven we ook een criterium voor confluentie.

Door alle clauses te laten gelden als definitionele gelijkheden, gedragen definities aan de hand van pattern matching zich meer als wiskundige definities, in plaats van programma-instructies. We verliezen echter de mogelijkheid om pattern matching te vertalen naar het toepassen van eliminatoren, wat de theoretische studie bemoeilijkt. Om dit verlies te compenseren geven we een theoretisch resultaat dat ons een equivalentie geeft met niet-overlappende definities.

Samenvatting voor niet-specialisten (Dutch)

Software speelt steeds meer een centrale rol in onze samenleving. We vinden het terug op onze gsm, op onze tv en in onze auto; maar ook in telefooncentrales, medische apparatuur, ruimtevaartuigen en militaire apparatuur. Voor zulke toepassingen is het van levensbelang dat de software foutloos werkt. Daarom wordt deze software op voorhand grondig getest. En toch leert de praktijk ons keer op keer dat er altijd fouten in software blijven staan.

Is het dan onmogelijk om een softwareprogramma te maken dat steeds doet wat we er van verwachten? De enige manier om honderd procent zeker te zijn, is door wiskundig te bewijzen dat het programma correct werkt. Maar wiskundige bewijzen geven is moeilijk, en daardoor alleen weggelegd voor specialisten. Bovendien bevatten zelfs de bewijzen van specialisten soms nog fouten, waardoor het hele bewijs waardeloos kan worden. Toch is dit al een stap in de goeie richting.

Om het gemakkelijker te maken om wiskundige bewijzen over programma's te geven, zijn er nieuwe programmeertalen ontworpen. In deze programmeertalen kunnen we niet alleen programma's schrijven, maar ook wiskundige bewijzen. De computer kijkt deze bewijzen dan automatisch na. Als er fouten in staan, weigert de computer om het programma uit te voeren. Door deze strenge controle kunnen we eindelijk zeker zijn dat het programma correct werkt.

Jammer genoeg zijn deze programmeertalen momenteel niet bruikbaar zonder een grondige kennis van de onderliggende wiskundige theorie. Het is echter niet realistisch om deze kennis te verwachten van iemand die 'gewoon' een correct programma wil schrijven. Dus een belangrijk doel is om deze talen zo te ontwerpen dat de benodigde kennis tot een minimum wordt beperkt.

Een voorbeeld van een veelgebruikte techniek om programma's te schrijven is gevalsonderscheid. Dit is een manier om aan de computer te zeggen: in dit geval moet je zus doen, in dat geval zo, enzovoort. De technische naam hiervoor is pattern matching. Als we iets willen bewijzen over een programma dat gevalsonderscheid gebruikt, moeten we dit voor elk geval afzonderlijk bewijzen. Een bewijs over een programma met gevalsonderscheid maakt dus gebruik van hetzelfde gevalsonderscheid!

In huidige programmeertalen ontstaat er een probleem wanneer er meerdere gevallen van het gevalsonderscheid tegelijkertijd van toepassing zijn. In deze situatie wordt er namelijk gekozen voor het eerste geval dat van toepassing is. Dit maakt het moeilijker om het bewijs te geven voor de andere gevallen die ook van toepassing waren.

Het doel van mijn thesis is om gevalsonderscheid gemakkelijker bruikbaar te maken. Dit doe ik door niet te kiezen voor het eerste geval dat van toepassing is, maar voor alle toepasbare gevallen tegelijk. Het is natuurlijk niet realistisch of wenselijk om ook echt alle gevallen tegelijk uit te voeren. In plaats daarvan laat ik de computer controleren of de programma's voor de verschillende gevallen samenvloeiend zijn. Dit wil zeggen dat alle gevallen hetzelfde resultaat geven. In de praktijk wordt dus toch slechts een geval gekozen, maar we weten nu wel dat de andere gevallen hetzelfde resultaat zouden geven.

Programmeertalen waarin we ook bewijzen kunnen geven staan nog ver van het punt waarop ze door iedereen bruikbaar zijn. Hopelijk brengt deze thesis dat punt weer een klein beetje dichterbij. En misschien komt er ooit een dag waarop het even gemakkelijk is om een programma te schrijven, als om te bewijzen dat het correct is.

Acknowledgements

Thank you to my supervisors Frank and Dominique, for allowing me to work on this fascinating subject and for allowing me to do whatever I thought was best, but always being ready with advice when I needed it.

Thank you to the Agda developers for their continuing work on this amazing language, and thank you to the Agda community for providing many interesting discussions, which provided the basis for this thesis.

Thank you to Rita for providing valuable advice on the formatting of the references.

Thank you to all my friends and fellow students for providing distraction when it was needed most.

Thank you to my father for listening to me whenever I was stuck on a problem and proofreading preliminary versions of this text.

And finally, thank you to both my parents for their support and love they gave while I was working on this thesis, and all the times before.

Contents

Chapter 1

Introduction

Pattern matching is a useful mechanism to define functions in functional programming languages. Definitions by pattern matching are given by a set of equalities called *clauses*. An example of a definition by pattern matching is the addition `plus` on natural numbers

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

Here `zero` represents the number 0 and `suc` m the number $m + 1$. This example already shows two powerful features of pattern matching: *case distinction* and *recursion*.

Languages based on type theory such as Agda or Coq are similar to functional languages, only they have more expressive types. These types are called *dependent types* because they can depend on values. Dependent types allow us to formulate arbitrary logical propositions as types. To prove a proposition formulated as a type, we just have to give a program of that type. For example, here is a program called `lemma`, with one argument m, which proves that `plus` m `zero` $= m$ (remark that this is not immediately obvious from the definition of `plus`):

$$\begin{array}{llcl} \texttt{lemma} & \texttt{zero} & = & \texttt{refl}\ \texttt{zero} \\ \texttt{lemma} & (\texttt{suc}\ m) & = & \texttt{cong}\ \texttt{suc}\ (\texttt{lemma}\ m) \end{array}$$

Here `refl` m is a proof that $m = m$ and `cong` $f\ p$ is a proof that $f\ x = f\ y$ if p is a proof that $x = y$. Remark that this program uses pattern matching to give a proof by *case distinction* and *induction*. So in a language with dependent types, pattern matching becomes even more powerful because it allows us to give not just programs, but also proofs.

With great power comes great responsibility, as the saying goes. If we are not careful, it is very 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 dependently typed languages will put certain restrictions on definitions by pattern matching:

- To ensure completeness, it is required that the patterns of the definition form a *covering*.
- To ensure termination, it is required that the definition is *structurally recursive*.

If these restrictions are satisfied, it is possible to translate the definition to one that doesn't use pattern matching, but only the (from a theoretical perspective) simpler *eliminators* [GMM06]. This translation ensures that definitions satisfying these restrictions are correct. However, these restrictions also limit the expressiveness for the programmer.

1.1 Goal of this thesis

In the dependently typed language Agda, it is allowed to give more general definitions by pattern matching. Internally these definitions are translated to a standard form that satisfies the above requirements. Hence this translation allows high expressiveness while keeping it doable to check correctness. However, some details of the definition are lost in the translation:

- Overlapping patterns are interpreted on a first-match basis, so some of the information in the later patterns can be lost.
- Not all clauses of the definition are preserved as definitional equalities in the translation, even if the patterns do not overlap.
- The order of the clauses in the definition influences the result of the translation, even if the patterns do not overlap.

This results in discrepancies between the definition given by the programmer and the definition used internally by Agda, and confusion ensues.

In this thesis, we will describe a new interpretation of pattern matching that treats all clauses as definitional equalities, even when the patterns overlap. For example, this will allow us to define `plus` as follows:

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

By defining `plus` like this, some facts such as $\texttt{plus}\ m\ \texttt{zero} = m$ don't need to be proven anymore, and some others become easier to prove. By interpreting clauses directly as definitional equalities, we eliminate the translation step. This has the following consequences:

- Overlapping patterns do not follow the first-match semantics any more.
- Each clause in the definition is a *definitional equality*, i.e. it has a clear interpretation that doesn't depend on the other clauses.
- In particular, the order in which the clauses are given doesn't influence their interpretation.

This generalized form of pattern matching has been implemented as a modification to Agda.

In order to use definitions with overlapping patterns, we need to be able to check their correctness automatically. In particular, we need to check whether a defined function will

always give the same result for the same arguments. This property is called *confluence*. We will describe an algorithm that automatically checks whether a given definition is confluent. This algorithm has also been implemented as part of the modification to Agda.

1.2 Related work

It would not have been possible to write this thesis without standing on the shoulders of the giants. My two main sources of inspiration were the paper on eliminating dependent pattern matching by H. Goguen, C. McBride, and J. McKinna [GMM06] and U. Norrell's PhD thesis on Agda [Nor07]. A special mention goes to a paper by A. Gräf [Gr91] for inspiring a lot of the ideas about overlapping patterns in this thesis, even if it is not always visible in the final version.

When I started this thesis, I hardly knew anything about type theory. Of course there are many great introductions to the subject, for me the books by S. Thompson [Tho99] and B. Nordström et al. [NPS90] were especially helpful. The description of type theory that is used in this thesis is the Unified Theory of Dependent Types (UTT) of Z. Luo in [Luo94] which is also studied in detail by H. Goguen in [Gog94].

On the subject of dependent pattern matching, the paper by T. Coquand [Coq92] that started it all should certainly be mentioned. Other useful references are [McB00] by C. McBride and [MM04] by C. McBride and J. McKinna.

1.3 Overview

Chapter 2 gives a short introduction to type theory with dependent types, in particular the theory of UTT (Unified Theory of dependent Types) [Luo94]. We will pay special attention to *inductive families of dependent types*, because they play an important role in dependent pattern matching. We will also discuss the fundamental properties of type theory: subject reduction, strong normalization, Church-Rosser, and completeness of β-evaluation. These are the properties that need to be preserved whenever we make any extension to type theory, such as ours.

Chapter 3 gives a general description of pattern matching, first in the case of simple types and then extending it to the dependent case. We will see that pattern matching on inductive families requires a new kind of patterns called *inaccessible patterns*. We will also give the three conditions that a definition by pattern matching has to satisfy in order not to break the fundamental properties: completeness, termination, and confluence.

Chapter 4 describes the algorithms that are used in dependently typed languages to automatically check these conditions. In particular, we will define *coverings* and the *structural order* which can be used to check completeness and termination respectively. We end the chapter with a description of case trees, which can be used to efficiently represent definitions by pattern matching. These case trees are exactly the internal representations of definitions by pattern matching used by Agda.

Chapter 5 gives some deficiencies of the algorithms described in chapter 4. Because they require patterns to form a covering, they do not allow overlapping patterns, and the result depends on the order of the patterns. To fix these problems, we will give a new interpretation of pattern matching that doesn't require patterns to form a covering. In particular, we allow the patterns in a definition to overlap.

Chapter 6 describes how it can be checked whether definitions with overlapping patterns are correct. We will describe algorithms for completeness and confluence checking. We will also describe two equivalent ways in which definitions with overlapping clauses can be represented as case trees.

Chapter 7 gives some examples of definitions with overlapping patterns. We will also compare our solution to an alternative one that requires explicit proofs of confluence.

Chapter 8 finally describes a theoretical result that gives an extensional equivalence between definitions with overlapping patterns and definitions without.

Appendix A shows how these modifications take on a concrete form in Agda. It also gives a few details about the implementation which are specific to Agda.

Chapter 2

Type theory

Type theory is a formal language that is used by both computer scientists and mathematicians. For computer scientists, type theory is useful because it brings programming and logic together in one language. This allows properties of programs to be proven in the language of the program itself. For mathematicians, type theory is an alternative to classical logic and set theory. Each proof in type theory is a constructive proof: proofs are programs that can carry out the concerned constructions. Because type theory is used in both worlds, new ideas in one can be used directly in the other. This two-way interaction is precisely what makes type theory so interesting.

The first two sections of this chapter give an informal introduction to type theory, first in general and then to dependent types specifically. Readers who are familiar with type theory can skip to section 2.3, which gives a more formal description. The next section then introduces inductive families, without which pattern matching would not be possible. Finally, the last section describes the most important meta-theoretical properties of type theory.

2.1 What is type theory?

Formally, type theory is described by a set of rules that can be used to deduce the true judgments of the system. As a formal system, type theory doesn't give any interpretation to these judgments, but it can be interpreted in several ways. A logician can use type theory as a *formal logic*, a computer scientist can use type theory as a *functional programming language*, and a mathematician can use type theory as a *framework for constructive mathematics*. The central concept in type theory are judgments of the form

$$p : P$$

Depending on our viewpoint coming from logic, computer science or mathematics, this can be read as

> "p is a proof of proposition P"
>
> *or* "p is a term of type P"
>
> *or* "p is an element of the set P".

Under these interpretations a proposition corresponds to a type, and a proof of this proposition corresponds to a term of this type. A false proposition in particular corresponds to an empty type, i.e. a type without terms. This correspondence between logic and functional programming is often called the *Curry-Howard correspondence*. We will use the words 'type' and 'proposition', as well as 'term' and 'proof' interchangeably, depending on the interpretation we wish to stress.

Each interpretation of type theory will give motivation to add new types to the theory. For example, we will introduce types that correspond to the qualifiers $\forall$ and $\exists$ from predicate logic. Using the Curry-Howard correspondence, we will also be able to write programs using these new types. These new types are called *dependent types* and will give a more expressive type system than in most common programming languages.

2.1.1 Propositional logic in type theory

To express logical formulas in type theory, we need some logical connectives such as conjunction $\wedge$, disjunction $\vee$, implication $\to$ and negation $\neg$. We can use these connectives to build new propositions from existing ones. Contrary to classical logic, these connectives are not defined by a truth table, but instead by the set of their possible proofs.

We will represent a proof as a piece of *data* (a term) that can be used to reconstruct the proof. The form of this data depends on the proposition we wish to prove: (Here, A and B are two arbitrary propositions)

$A \wedge B$ To prove $A \wedge B$, it is sufficient to prove A and to prove B. Hence we can represent a proof of $A \wedge B$ as a pair (a, b) where a is a proof of A and b is a proof of B. Conversely, if we have a proof p of $A \wedge B$, this gives us proofs of A and B. We denote these as `fst` p and `snd` p respectively.

$A \vee B$ To prove $A \vee B$, it is sufficient to either prove A or prove B. Hence we can represent a proof of $A \vee B$ as a term `inl` a where a is a proof of A or `inr` b where b is a proof of B. Afterwards, we can analyze the structure of a proof of $A \vee B$ to see whether it is of the form `inl` a or `inr` b. This is a first example of *pattern matching*, which will be discussed later.

$A \to B$ To prove $A \to B$, we must be able to transform an arbitrary proof of A into a proof of B. Hence a proof of $A \to B$ is a description of a *program* that performs this transformation. If we are given a proof a of A and a proof f of $A \to B$, we can apply f to a to get a proof $f\ a$ of B.

$\top$ The trivial proposition $\top$ has one proof that we denote by `tt`.

$\bot$ The absurd proposition $\bot$ has no proof. If we would have a proof b of $\bot$, then we could construct a proof $\texttt{absurd}_A\ b$ of any proposition A.

$\neg A$ We define $\neg A$ to be equal to $A \to \bot$, i.e. a proof of $\neg A$ is a program that transforms a proof of A into a proof of $\bot$.

We defined a proof of $A \to B$ as a *program* that transforms proofs, which are represented as data. What do we mean when we say 'a program'? To give a real proof of $A \to B$, a program must ...

... be defined for each a of type A,

... never get stuck,

... give a result after a finite number of steps,

... give a valid proof of B as a result,

... always give the same result for the same input.

We will represent this kind of proof-transforming programs by lambda expressions. A lambda expression of type $A \to B$ is of the form $\lambda(x : A).\ b$ where $b : B$ and x is a variable of type A that can occur freely in b. To apply this expression to a proof a of A, we must substitute a for x in b. The result of this substitution is written as $b[x \mapsto a]$. Thus a program in type theory is a function, represented as a term in the form of a lambda expression.

As an example, we will give a proof of $(((A \vee B) \to C) \wedge A) \to C$. To do this, we must transform a proof of $((A \vee B) \to C) \wedge A$ to a proof of C. A proof of $((A \vee B) \to C) \wedge A$ has two components; the first component is a proof $\texttt{fst}\ p$ of $(A \vee B) \to C$ and the second component is a proof $\texttt{snd}\ p$ of A. We can now construct a proof $\texttt{inl}\ (\texttt{snd}\ p)$ of $A \vee B$ and hence we can apply the function $\texttt{fst}\ p$ to this to get a proof of C. The full proof of $(((A \vee B) \to C) \wedge A) \to C$ looks like this:

$$\lambda(p : ((A \vee B) \to C) \wedge A)).\ (\texttt{fst}\ p)\ (\texttt{inl}\ (\texttt{snd}\ p)$$

2.1.2 Type theory as a programming language

If we regard the propositions and proofs we have defined as terms and types instead, we already have a (primitive) programming language: $A \to B$ is a type of *functions*, $A \wedge B$ is a type of *pairs* and $A \vee B$ is the *disjoint union* of A and B. In a programming context, we denote $A \wedge B$ as the *product* $A \times B$ and $A \vee B$ as the *sum* $A + B$ of the two types A and B. This language with $\to$, $\times$ and $+$ is called the *simply typed lambda calculus*.

To write real programs, we need some basic types such as booleans or natural numbers.

`Bool` We define a new type `Bool` with two canonical terms $\texttt{true} : \texttt{Bool}$ and $\texttt{false} : \texttt{Bool}$. For any type A with terms $x, y : A$ and a boolean $b : \texttt{Bool}$ we define the term $\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y$, which evaluates to x if $b = \texttt{true}$ or to y if $b = \texttt{false}$. This allows us to define functions by case distinction.

`Nat` We define a new type `Nat` with one base term $\texttt{zero} : \texttt{Nat}$ and a successor function $\texttt{suc} : \texttt{Nat} \to \texttt{Nat}$. For example, the number 2 can be represented as $\texttt{suc}\ (\texttt{suc}\ \texttt{zero})$. For any type A with a term $z : A$ and a function $s : A \to A$ and a natural number $n : \texttt{Nat}$ we define $\texttt{natrec}\ n\ z\ s : A$, where $\texttt{natrec}\ n\ z\ s$ equals $s\ (s\ \ldots (s\ z))$ (s applied n times to z). This allows us to define functions by natural induction.

For example, we can define the function $\texttt{iszero} : Nat \to Bool$ as follows:

$$\texttt{iszero} = \lambda n.\ \texttt{natrec}\ n\ \texttt{true}\ (\lambda m.\ \texttt{false})$$

To use type theory as a programming language, we need a concept of computation on terms. This is done by giving *evaluation rules*, for example $\texttt{fst}\ (a, b) \longrightarrow a$. Figure 2.1 gives an overview of the evaluation rules for the terms we have considered so far.

$$(\lambda(a : A).\ b)\ a' \longrightarrow b[a \mapsto a']$$

$$\texttt{fst}\ (a, b) \longrightarrow a \qquad \texttt{snd}\ (a, b) \longrightarrow b$$

$$\texttt{if true then}\ x\ \texttt{else}\ y \longrightarrow x \qquad \texttt{if false then}\ x\ \texttt{else}\ y \longrightarrow y$$

$$\texttt{natrec zero}\ z\ s \longrightarrow z \qquad \texttt{natrec}\ (\texttt{suc}\ n)\ z\ s \longrightarrow s\ (\texttt{natrec}\ n\ z\ s)$$

Figure 2.1: Evaluation rules for the simply typed lambda-calculus.

Together with these evaluation rules there are a number of congruence rules that allow us to apply the evaluation rules to arbitrary subterms. For example, if $t_1 \longrightarrow t_1'$ and $t_2 \longrightarrow t_2'$, then we also have $t_1\ t_2 \longrightarrow t_1'\ t_2$ and $t_1'\ t_2 \longrightarrow t_1'\ t_2'$. Together these rules define a binary relation $\longrightarrow$ on the set of all terms. The reflexive and transitive closure of this relation is written as $\longrightarrow^*$, so we have $t \longrightarrow^* s$ if and only if there exist terms $t_0, \ldots, t_n$ such that $t = t_0 \longrightarrow \ldots \longrightarrow t_n = s$.

A term t is in *normal form* if there exists no term s such that $t \longrightarrow s$. If $t \longrightarrow^* s$ with s in normal form, then we say that s is a normal form of t. A normal form of a term is (in a sense) the simplest form of this term, so it is often seen as giving the meaning of the term.

A term in which all variables are bound by lambda expressions is called a *closed term*. A very important property of type theory is that every closed term has a unique normal form. It follows that we can check the equality of two terms by evaluating them to normal form and comparing their normal forms. These and other meta-theoretical properties of type theory are discussed in more detail in section 2.5.

2.1.3 Constructive mathematics in type theory

Type theory can also be used as an alternative to set theory, where we interpret types as sets and terms as elements of that set. So far we have only constructed sets without internal structure. It is possible to add extra structure to a type, for example that of a group, a topological space, a category, ... This allows us to formulate and prove propositions about those objects, and have these proofs automatically checked by the computer. Moreover, the proofs we give have a computational content, in the sense that a proof of "there exists an x such that y" contains a program that can construct the x mentioned in the theorem. Very recently, a deep theorem from group theory called the Odd Order Theorem, which states that every finite group of odd order is solvable, has been fully formalized in type theory using the Coq proof assistant [Gon13].

As an example of how this extra structure on a type can be defined, we can take two points $p_1, p_2 : P$ and define a new type $p_1 \rightsquigarrow p_2$ of *paths* from p_1 to p_2. These types are studied in a rich and active area called *homotopy type theory*. A good introduction to this fascinating subject can be found in [Rij12].

2.2 What are dependent types?

From the logic perspective, the types we have introduced so far correspond to propositional logic. We will now introduce new types that correspond to predicate logic. For example,

we want to be able to express and prove the following proposition:

"For each natural number n, `suc` n is not equal to `zero`."

For this, we need types that depend on a value. For example, we can define a type `IsZero` n, which depends on the value of the natural number n. The only term of a type of the form `IsZero` n is the term `iszero` of type `IsZero zero`. This term `iszero` is called a *constructor*. So for each natural number n we have a concrete type `IsZero` n, which is non-empty if and only if n is equal to `zero`.

Another example of a dependent type is `Square` n that expresses the proposition "n is a perfect square". The terms are of the form `sq` n, which has type `Square` n^2. Hence for all m : `Nat` for which m is not a perfect square, `Square` m is an empty type, i.e. a proposition without proof.

In general, a dependent type is a *family of types* indexed by a base type. Some types of the family might be empty, others might be not. Hence a dependent type expresses a certain property the terms of the base type may or may not have.

Dependent types are not just used for formulating propositions, but also for programming. For example, we can define a dependent type `Fin` n that has exactly n elements for each n : `Nat`. This is a dependent type over the base type `Nat`. This type is useful for indexing a list of length n, because it guarantees us that indices which are too big will never be used.

2.2.1 The dependent product

Suppose we have a dependent type $D\ a$ over a base type A. We introduce a new type $\forall(a : A).\ D\ a$ that expresses that $D\ a$ is inhabited (nonempty) for all $a : A$. This type is called a *dependent product*. A term of this type is a program that gives a term of type $D\ a$ on input A, i.e. a lambda expression $\lambda(x : A).\ p$ where $p : D\ a$ can contain free occurrences of the variable x. For example, we can translate the proposition "For each natural number n, `suc` n is not equal to `zero`" to type theory as the type:

$$\forall(n : \texttt{Nat}).\ \neg(\texttt{IsZero}\ (\texttt{suc}\ n))$$

Remark that $\forall(a : A).\ D\ a$ is a generalization of the type $A \to B$ where the simple type B has been replaced by a dependent type $D\ a$. For this reason we often write $(a : A) \to D\ a$ instead of $\forall(a : A).\ D\ a$. The terms of this type are functions of which the type of the result depends on the value of the concrete argument. In contrast to the regular function type $A \to B$, this is a feature that does not occur in most typed programming languages.

We can also interpret $\forall(a : A).\ D\ a$ as the (possibly infinite) product of all types $D\ a$ for $a : A$. A term $t : \forall(a : A).\ D\ a$ can be seen as a tuple and its component at index a is given by $t\ a$. This explains why $\forall(a : A).\ D\ a$ is called the dependent product.

2.2.2 The dependent sum

Analogously to the dependent product, we also want a type expressing that $D\ a$ is inhabited for at least one $a : A$. So we define the type $\exists(a : A).\ D\ a$, which is called a *dependent sum*. A term of type $\exists(a : A).\ D\ a$ is given by a pair (a, p) where a has type A and p has type $D\ a$. So to give a proof of $\exists(a : A).\ D\ a$, we have to construct an actual term a

satisfying $D\ a$, and this term can be reconstructed from the proof of $\exists(a : A).\ D\ a$. This is not possible in classical logic, and that is one of the reasons why type theory is called a constructive logic.

On one hand, we can see $\exists(a : A).D\ a$ as a generalization of the regular product $A \times B$ where the type of the second component depends on the value of the first component.

On the other hand, we can see $\exists(a : A).\ D\ a$ as a disjoint union of the family of all types $D\ a$. In this case the first component is only a label that assures that the union is disjoint. For example, $\exists(n : \texttt{Nat}).\ (\texttt{Fin}\ n)$ is the disjoint (infinite) union of all the finite types $\texttt{Fin}\ n$.

A third possible interpretation of $\exists(a : A).\ D\ a$ is as a subtype of A consisting of all a for which $D\ a$ is inhabited. For example, $\exists(n : \texttt{Nat}).\ (\texttt{Square}\ n)$ is the subtype of all natural numbers which are a perfect square.

2.2.3 The identity type

Another important proposition we want to express in logic is equality between two objects. To do this, we introduce for each type A and terms $x, y : A$ an *identity type* $x \equiv_A y$, which is inhabited if x and y evaluate to the same expression. The only canonical term of this type is $\texttt{refl}\ x$, which has type $x \equiv_A x$ for any $x : A$. This is called `refl` because it expresses the reflexivity of the relation $\equiv_A$.

We know how to construct a term of type $x \equiv_A x$, but what can we do with a given term of type $a \equiv_A b$? If we have a term p of type $a \equiv_A b$, then we know that a and b are equal, hence we should be able to substitute b for a in any term. This is essentially Leibniz' law that says that equals can be substituted for equals. Let $C\ x$ be a dependent type over the base type A, then we denote this substitution as $\texttt{J}\ b\ p : C\ a \to C\ b$. If p is of the form $\texttt{refl}\ a$, then b must be equal to a by definition of `refl` and hence we have the evaluation rule $\texttt{J}\ a\ (\texttt{refl}\ a)\ c \longrightarrow c$. Remark that we don't have $\texttt{J}\ a\ p\ c \longrightarrow c$ for arbitrary p since this would not be type-correct.

This substitution rule can be generalized so that the type C is not only dependent over $a : A$ but also over $p : a \equiv_A b$. Let A be a type, $a : A$, and C be a dependent type indexed over $x : A$ and $p : a \equiv_A x$. The full type of J then becomes:

$$\texttt{J} : (b : A) \to (p : a \equiv_A b) \to C\ a\ (\texttt{refl}\ a) \to C\ b\ p \tag{2.1}$$

By using this substitution rule it is possible to give terms

$$\texttt{sym} : (x : A) \to (y : A) \to x \equiv_A y \to y \equiv_A x \tag{2.2}$$

and

$$\texttt{trans} : (x : A) \to (y : A) \to (z : A) \to x \equiv_A y \to y \equiv_A z \to x \equiv_A z \tag{2.3}$$

that prove that the relation $\equiv_A$ is symmetric and transitive (see for example [Rij12] p.15-16). In other words, $\equiv_A$ is an equivalence relation on the terms of type A.

It is surprising that with only this substitution rule it is impossible to prove that $\texttt{refl}\ x$ is the only proof of $x \equiv_A x$. In other words, it is impossible to give a term of type $(p : x \equiv_A x) \to p \equiv_{x \equiv_A x} (\texttt{refl}\ x)$. In homotopy type theory [Rij12], $x \equiv_A y$ is interpreted as a type of paths from x to y, where $\texttt{refl}\ x$ is the trivial path; standing still at x. Hence the possibility of other terms of type $x \equiv_A y$ is essential for the development of homotopy type theory.

2.2.4 Universes

We can regard a dependent type over a base type A as a function from A to types. To formalize this intuition, we need to introduce *universes*. A universe is a type of which the terms are types themselves. The archetypical example of a universe is the universe Set, which contains all base types we have seen so far: `Bool`, `Nat`, $\top$, $\bot$, and all types we can form by starting from these and using $\wedge$, $\vee$, $\to$, $\forall$, $\exists$, and $\equiv$. The types in the universe Set are called the *small types*.

A dependent type over a base type A can be seen as a function of type $A \to Set$; for example `IsZero` is a function of type `Nat` $\to Set$. The symbols $\forall$ and $\exists$ are in fact themselves terms of type $(A : Set) \to (A \to Set) \to Set$, and the identity type has type $(A : Set) \to (a : A) \to (b : A) \to Set$.

So if Set is itself a type (of types), can we say that $Set : Set$? It turns out that it is not possible to do this in a consistent way, because doing so leads to problems similar to Russell's paradox from set theory. We will instead introduce a hierarchy of universes $Set_0 = Set, Set_1, Set_2, \ldots$ where $Set_0 : Set_1$, $Set_1 : Set_2$, etc. Each of these universes us closed under type constructors such as $\forall$ and $\exists$, as in the definition of Set.

2.3 Formal description

This section formally describes the core type theory we will use. It is based on UTT (Unified Theory of Dependent Types) [Luo94]. This version of type theory only contains (dependent) function types and universes. All remaining types can be defined using inductive families, as described in the next section. The original description is made in a meta-level logical framework. Since we use UTT more as a tool here and not as the subject of study, we will not be quite so formal here. UTT also contains a separate impredicative universe of propositions, which we also omit because it is not needed for pattern matching.

2.3.1 The building blocks

Contexts

When formally describing type theory, we want in general be able to say that a term t has a certain type T under a set of assumptions on the types of the free variables in t. We call such a set of assumptions a *context*. We write a context as a list of variables with a type annotation

$$(x_1 : X_1)(x_2 : X_2) \ldots (x_n : X_n)$$

where, in general, the variables $x_1, \ldots, x_{i-1}$ can occur in the type X_i. This allows the type of an assumption to depend on a previous assumption, for example in the context $(n : \texttt{Nat})(s : \texttt{Square}\ n)$. Another example of a context is $(A : Set)(x : A)(y : A)(p : x \equiv_A y)$. An arbitrary context is usually denoted by a greek capital letter Γ, Δ, ... In order to avoid inconsistencies, we allow each variable to occur only once in each context.

Contexts are multifunctional: we can also use them as the type of a *list* of terms. We extend the typing relation $x : T$ to a relation between lists of terms and contexts as follows: $t_1\ \ t_2\ \ \ldots\ \ t_n : (x_1 : X_1)(x_2 : X_2) \ldots (x_n : X_n)$ if and only if $t_1 : X_1$ and

$$
\begin{array}{rll}
\mathit{var} & = x \mid y \mid z \mid \ldots & \\
\mathit{term} & = \mathit{var} & \text{variable} \\
& \mid \lambda(\mathit{binding}).\, \mathit{term} & \text{lambda abstraction} \\
& \mid \mathit{term}\ \mathit{term} & \text{application} \\
& \mid (\mathit{binding}) \to \mathit{term} & \text{dependent product} \\
& \mid \mathit{Set}_0 \mid \mathit{Set}_1 \mid \mathit{Set}_2 \mid \ldots & \text{universes} \\
\mathit{binding} & = \mathit{var} : \mathit{term} & \\
\mathit{context} & = \epsilon & \text{empty context} \\
& \mid \mathit{context}(\mathit{binding}) & \text{extended context}
\end{array}
$$

Figure 2.2: Syntax of UTT

$t_2\ \ldots\ t_n : (x_2 : X_2') \ldots (x_n : X_n')$, where $X_i' = X_i[x_1 \mapsto t_1]$. For example, we have

$$9\ (\texttt{sq}\ 3) : (n : \texttt{Nat})(s : \texttt{Square}\ n)$$

and

$$\texttt{Bool}\ \texttt{true}\ \texttt{true}\ (\texttt{refl}\ \texttt{true}) : (A : \mathit{Set})(x : A)(y : A)(p : x \equiv_A y)$$

If we have a context $\Gamma = (x_1 : X_1)(x_2 : X_2) \ldots (x_n : X_n)$ and B is a type with free variables $x_1, \ldots, x_n$, then we can form the iterated dependent product $(x_1 : X_1) \to (x_2 : X_2) \to \ldots \to (x_n : X_n) \to B$. Because it quickly becomes tedious to write all these arrows, we will write this type as $(x_1 : X_1)(x_2 : X_2) \ldots (x_n : X_n) \to B$ or simply $\Gamma \to B$. This notation for the iterated dependent product is called the *telescope notation*.

We will also sometimes use a context to stand for the list of variables declared in the context. For example, if $f : \Gamma \to T$ then we have $\Gamma \vdash f\ \Gamma : T$.

Syntax

In figure 2.2, we give the formal syntax of (our version of) UTT. Remark that terms and types both belong to the same syntactic class *term*. This is our way to allow types to be terms themselves (i.e. types can be part of a universe). The types are simply those terms that have type Set_i for some i.

Another way to work with universes, which is used in the original description of UTT, is to define for each term t in a universe U a type $\mathit{El}(t)$. This has the advantage that there is a clear syntactic distinction between terms and types, but we believe our approach results in a simpler and more uniform theory.

Definition 2.1 (Bound occurrence, closed term). *All occurrences of a variable x in an expression of the form $\lambda(x : A).\, t$, $(x : A) \to T$ or $(x : A), \Gamma$ are* bound.

Definition 2.2 (Free variables). *If an occurrence of a variable is not bound, it is called* free. *The set of all variables that occur freely in an expression e is denoted as* $FV(e)$

Definition 2.3 (Closed/open term). *If a term has no free variables, then we call the term* closed, *otherwise it is* open.

Strictly speaking, it only makes sense to say that a particular occurrence of a variable is free or bound. But if it is clear which occurrence of the variable is meant (e.g. because there is only one, or because it doesn't matter which one), then we say that the variable itself is free or bound respectively.

If two terms have the same syntactic structure, then clearly it makes no sense to distinguish between them. But if two terms only differ in the names of their bound variables - for example $\lambda(x : Set_0).\ x$ and $\lambda(y : Set_0).\ y$ - then we still regard these as having the same meaning. This is expressed in definition 2.4.

Definition 2.4 (Syntactic equality). *If two terms have the same syntax up to renaming of bound variables, we call them* syntactically equal.

We will never distinguish between two syntactically equal terms.

Substitutions

We denote the *simultaneous substitution* of the terms $t_1, \ldots, t_n$ for the variables $x_1, \ldots, x_n$ as $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$. We require that $t_i \neq x_i$ to avoid pathological cases. In particular, the identity substitution is denoted as []. The application $t[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n$ is defined as the term t where the free occurrences of $x_1, \ldots, x_n$ have been replaced by $t_1, \ldots, t_n$ respectively. We assume substitutions rename bound variables as needed to avoid variable capture. This renaming is always possible because we assume an infinite supply of fresh new variables and because our definition of syntactic equality allows renaming of bound variables.

The *domain* of a substitution $\sigma = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ is the set of variables $dom(\sigma) = \{x_1, \ldots, x_n\}$. If we have two substitutions σ and τ with disjoint domains, then we can *compose* these two substitutions. This is written as $\sigma;\tau$ (first σ, then τ) and is done by applying τ to all terms of σ, and then adding the two lists together:

$$\begin{aligned}&[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]; [y_1 \mapsto s_1, \ldots, y_m \mapsto s_m] \\ &\quad = [x_1 \mapsto t_1\tau, \ldots, x_n \mapsto t_n\tau, y_1 \mapsto s_1, \ldots, y_m \mapsto s_m] \\ &\quad (\text{where } \tau = [y_1 \mapsto s_1, \ldots, y_m \mapsto s_m])\end{aligned} \tag{2.4}$$

Remark that $t(\sigma;\tau)$ is syntactically equal to $(t\sigma)\tau$ for all terms t, as would be expected from a composition.

If Γ is a context and $\bar{t}$ is a list of terms such that $\bar{t} : \Gamma$, then we denote $[\Gamma \mapsto \bar{t}]$ for the substitution that maps the variables in Γ to their corresponding terms in $\bar{t}$.

Judgments

Judgments are the things we can say formally to describe properties of a formal system, in this case type theory. In our description of type theory, we will use the following three judgments:

Γ **valid** means that Γ is a valid context containing a number of hypotheses.

$\Gamma \vdash a : A$ means that we can deduce that a has type A from the hypotheses in the context Γ.

$\Gamma \vdash a = b : A$ means that we can deduce that a and b are equal terms of type A from the hypotheses in the context Γ.

Remark that these judgments are the *only* things we can formally say about our system. For example, the would-be judgment $\epsilon \vdash \texttt{zero} = \texttt{false}$ isn't true or false, it is simply meaningless because we can only make judgments about the equality of two terms if they have the same type.

2.3.2 Inference rules

To define a formal system such as type theory precisely, we use the system of natural deduction. In natural deduction, a formal system is described by a set of inference rules. An inference rule tells us how we can derive a conclusion from a set of hypotheses. An inference rule is written as

$$\frac{H_1 \qquad \dots \qquad H_n}{J}$$

where $H_1, \dots, H_n$ are the hypotheses and J is the conclusion. We can compose inference rules by replacing a hypothesis by another inference rule that has that hypothesis as a conclusion. By combining inference rules like this, we can build a *proof tree*. If there is a proof tree with in its leaves the judgments $H_1, \dots, H_n$ and at the root the judgment J, then we say that J is *derivable* from $H_1, \dots, H_n$.

Valid contexts

$$\frac{}{\epsilon \ \textbf{valid}}$$

$$\frac{\Gamma \vdash A : Set_i \qquad x \notin FV(\Gamma)}{\Gamma(x : A) \ \textbf{valid}}$$

Assumption rule

$$\frac{\Gamma \ \textbf{valid} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

General equality rules

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}$$

$$\frac{\Gamma \vdash t_1 = t_2 : A}{\Gamma \vdash t_2 = t_1 : A}$$

$$\frac{\Gamma \vdash t_1 = t_2 : A \qquad \Gamma \vdash t_2 = t_3 : A}{\Gamma \vdash t_1 = t_3 : A}$$

Subsumption rule

$$\frac{\Gamma \vdash t : A_1 \qquad \Gamma \vdash A_1 = A_2 : Set_i}{\Gamma \vdash t : A_2}$$

List of terms in a context

$$\frac{\Gamma \textbf{ valid}}{\Gamma \vdash \epsilon : \epsilon}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash \bar{t} : \Delta[x \mapsto t]}{\Gamma \vdash t\ \bar{t} : (x : A)\Delta}$$

$$\frac{\Gamma \vdash t = s : A \qquad \Gamma \vdash \bar{t} = \bar{s} : \Delta[x \mapsto t]}{\Gamma \vdash t\ \bar{t} = s\ \bar{s} : (x : A)\Delta}$$

Formation rule for Set_i

$$\frac{\Gamma \textbf{ valid}}{\Gamma \vdash Set_i : Set_{i+1}}$$

Formation rule for the dependent product

$$\frac{\Gamma \vdash A : Set_i \qquad \Gamma(x : A) \vdash B : Set_j}{\Gamma \vdash (x : A) \to B : Set_{\max(i,j)}}$$

$$\frac{\Gamma \vdash A_1 = A_2 : Set_i \qquad \Gamma(x : A_1) \vdash B_1 = B_2 : Set_j}{\Gamma \vdash (x : A_1) \to B_1 = (x : A_2) \to B_2 : Set_{\max(i,j)}}$$

Introduction rule for λ-abstractions

$$\frac{\Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda(x : A).\ t : (x : A) \to B}$$

$$\frac{\Gamma \vdash A_1 = A_2 : Set_i \qquad \Gamma(x : A_1) \vdash t_1 = t_2 : B}{\Gamma \vdash \lambda(x : A_1).\ t_1 = \lambda(x : A_2).\ t_2 : (x : A_1) \to B}$$

Application

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B[x \mapsto t]}$$

$$\frac{\Gamma \vdash f_1 = f_2 : (x : A) \to B \qquad \Gamma \vdash t_1 = t_2 : A}{\Gamma \vdash f_1\ t_1 = f_2\ t_2 : B[x \mapsto t_1]}$$

β-equivalence

$$\frac{\Gamma(x : A) \vdash t : B \qquad \Gamma \vdash s : A}{(\lambda(x : A).\ t)\ s = t[x \mapsto s] : B[x \mapsto s]}$$

η-equivalence

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad x \notin FV(f)}{\lambda(x : A).\ f\ x = f : (x : A) \to B}$$

2.3.3 Evaluation

The rules given in the previous section tell us which terms have which types and when two terms are equal, but they don't tell us anything about the computational meaning of a term. This computational meaning is given by an *evaluation rule*.

Definition 2.5 (Evaluation rule). *An* evaluation rule *is a binary relation $\longrightarrow$ on terms satisfying the following properties:*

- *If $s \longrightarrow t$, then $s\ u \longrightarrow t\ u$ for all u.*
- *If $s \longrightarrow t$, then $f\ s \longrightarrow f\ t$ for all f.*
- *If $s \longrightarrow t$, then $\lambda(x : A).\ s \longrightarrow \lambda(x : A).\ t$ for all A.*
- *If $A \longrightarrow B$, then $\lambda(x : A).\ t \longrightarrow \lambda(x : B).\ t$ for all t.*
- *If $A \longrightarrow B$, then $(x : A) \to C \longrightarrow (x : B) \longrightarrow C$ for all C.*
- *If $C \longrightarrow D$, then $(x : A) \to C \longrightarrow (x : A) \to D$ for all A.*

Definition 2.6 (β-evaluation). *The β-evaluation $\longrightarrow_\beta$ is the smallest evaluation rule that satisfies $(\lambda(x : A).\ t)\ s \longrightarrow_\beta t[x \mapsto s]$ for all s and t.*

Definition 2.7 (η-evaluation). *The η-evaluation $\longrightarrow_\eta$ is the smallest evaluation rule that satisfies $\lambda(x : A).\ f\ x \longrightarrow_\eta f$ whenever x is not free in f.*

For any evaluation rule $\longrightarrow$, we define $\longrightarrow^*$ as the reflexive and transitive closure of $\longrightarrow$.

Definition 2.8 (Normal form). *A term t is a* normal form *with respect to the evaluation rule $\longrightarrow$ if there is no term s such that $t \longrightarrow s$. A term t'* has normal form *t with respect to $\longrightarrow$ if $t' \longrightarrow^* t$ with t a normal form.*

We will usually drop the subscript in $\longrightarrow_\beta$ and $\longrightarrow_\eta$ and work with an evaluation rule $\longrightarrow$ that includes both β- and η-evaluation. Later on we will expand this evaluation rule for functions defined by pattern matching.

2.3.4 Equality in type theory

We have seen three different notions of equality between terms of type theory: syntactic equality, definitional equality, and propositional equality:

- Two terms s and t are syntactically equal if they look the same, i.e. if they have the same syntactical structure up to renaming of bound variables.
- Two terms s and t are definitionally equal (or *convertible*) if they are equal according to the evaluation rules, i.e. $\Gamma \vdash s = t : T$ where Γ is the context of free variables of s and t.
- Two terms s and t are propositionally equal if we can prove their equality in type theory, i.e. if there is a term p such that $\Gamma \vdash p : s \equiv_T t$ where Γ is the context of free variables of free variables of s and t.

When we write $s = t$ in an argument, we mean that s and t are definitionally equal (unless noted otherwise). For example, we can say that $(\lambda(x : A).\ x)\ a = a$.

When working with type theory in dependently typed languages such as Agda or Coq, it is more convenient to have definitional equalities rather than propositional equalities. This is because definitional equality can be checked automatically, while propositional equality has to be proven manually. This is especially true when reasoning about open terms.

The equalities are ordered from finest to coarsest: syntactic equality implies definitional equality implies propositional equality. It is clear that syntactic equality is strictly stronger than definitional equality: any two terms of the form $(\lambda(x : A).\ t)\ s$ and $t[x \mapsto s]$ are definitionally equal but not syntactically equal. Definitional equality is also strictly stronger than propositional equality: the terms `if` x `then true else true` and `true` are propositionally equal but not definitionally equal, since `if` x `then true else true` doesn't evaluate to `true`.

It can be startling that equality in type theory does not correspond to the intuitive notion of equality from set theory. This is because equality in set theory is *extensional*: two functions are equal if they give equal values for any closed argument. In contrast, equality in type theory is *intensional*: two functions are equal if they have the same definition.

The definitional equality is intensional. Indeed, there is no general way to derive $\Gamma \vdash f = g : A \to T$ from the fact that $\Gamma \vdash f\ a = g\ a : T$ for all closed terms a.

The propositional equality is also intensional because it is not always possible to construct a term of type $f \equiv_{A \to T} g$ from a collection of terms $p_a : f\ a \equiv_T g\ a$ for each closed $a : A$

Of course, the syntactic equality is extensional: if $f\ a$ and $g\ a$ are syntactically equal for all closed terms a, then f and g are syntactically equal. But the syntactic equality is too weak for most situations: it does not even satisfy simple equalities such as $(\lambda(x : A).\ x)\ a = a$.

2.4 Inductive families

An important aspect of type theory is the fact it is an open system: it is possible to add new types and functions to the theory, without invalidating older results. This allows

type theory to be used in many different ways, while staying small and elegant enough for theoretical study.

So far, we defined a number of types by hand. Each time we add a new type like this, it is possible it will make the whole system inconsistent as a logic. To avoid this, we have to check that the new type cannot give us more than we put into it. For example, for the type $A \wedge B$ we have a function $\texttt{fst} : A \wedge B \to A$. This is allowed because the only way to construct a term of type $A \wedge B$ involves giving an $a : A$. For the function `fst`, this was still quite easy to check, but in general this becomes harder.

To avoid inconsistencies, we will equip type theory with a general method to define new types. This method will always result in good types that do not break consistency. It is also general enough to define all types we have seen so far, with the exception of the function types $(a : A) \to B$ and the universes Set_i. We call them *inductively defined families of data types* or simply *inductive families*. Inductive families are discussed in detail in [Dyb94].

2.4.1 Examples

An inductive family is a dependent type defined by a set of *constructors*. Intuitively, constructors tell us how to build new canonical terms of the inductive family. The word 'inductive' reflects the fact that the constructors are essentially the *only* way to construct such terms. This fact will be exploited when we describe pattern matching in the next chapter. The archetypical example of an inductive family is the dependent type $\texttt{Vec}\ A\ n$ of vectors of A's of length n. There are two constructors for this family:

$$\epsilon : \texttt{Vec}\ A\ \texttt{zero}$$

is the empty vector of length 0 and

$$\texttt{cons} : (n : \texttt{Nat})(a : A)(v : \texttt{Vec}\ A\ n) \to \texttt{Vec}\ A\ (\texttt{suc}\ n)$$

allows us to add one element a in front of a vector v of length n in order to construct a vector of length $n + 1$.

Suppose that we want to define an inductive family D such that $\Gamma \vdash D : \Delta \to Set_i$. We call the variables in Γ the *parameters* of the family and the variables in Δ the *indices*. The difference between them is that the parameters are constant throughout the family, while the indices can vary from constructor to constructor. In the example of $\texttt{Vec}\ A\ n$, there is one parameter $A : Set$ and one index $n : \texttt{Nat}$. Each constructor c of D must have a type of the form $\Phi \to D\ \bar{\imath}$ where $\Gamma\Phi \vdash \bar{\imath} : \Delta$. For example, for the constructor ϵ we have $\Phi = \epsilon$ and $\bar{\imath} = \texttt{zero}$, while for `cons` we have $\Phi = (n : \texttt{Nat})(a : A)(v : \texttt{Vec}\ A\ n)$ and $\bar{\imath} = \texttt{suc}\ n$. Remark that the type D may occur in the context Φ. There is a restriction on where this may happen, this will be discussed in the next section.

We can define many of the types we saw before as inductive types:

`Nat` is an inductive family with no parameters, no indices, and two constructors $\texttt{zero} : \texttt{Nat}$ and $\texttt{suc} : \texttt{Nat} \to \texttt{Nat}$.

$A \wedge B$ is an inductive family with two parameters A and B, no indices, and one constructor $(\cdot, \cdot) : A \to B \to A \wedge B$.

$\exists(a : A).\ D\ a$ is an inductive family with two parameters A and D, no indices, and one constructor $(\cdot,\cdot) : (a : A) \to D\ a \to \exists(a : A).\ D\ a$. Remark that it is no problem for two different inductive families to share the same constructor name.

$\texttt{Square}\ n$ is an inductive family without parameters, with one index $n : \texttt{Nat}$, and one constructor $\texttt{sq} : (n : \texttt{Nat}) \to \texttt{Square}\ n^2$.

$x \equiv_A y$ is an inductive family with one parameter A, two indices $x, y : A$, and one constructor $\texttt{refl} : (x : A) \to x \equiv_A x$. It is also possible to see the first index x as a parameter instead, this doesn't change much to the meaning of the definition.

We can also use inductive families to define new properties and relations on existing types. For example, we can define an inductive family $n \leq m$ with indices $n, m : \texttt{Nat}$ and two constructors

$$\begin{aligned} \texttt{equal} &: (n : \texttt{Nat}) \to n \leq n \\ \texttt{smaller} &: (n : \texttt{Nat})(m : \texttt{Nat})(p : n \leq m) \to n \leq (suc\ m) \end{aligned}$$

2.4.2 Strict positiveness

In the definition of an inductive family D, it is allowed that D occurs in the type of the arguments of its constructors. For example, $\texttt{Nat}$ is the type of the argument of the constructor $\texttt{suc} : \texttt{Nat} \to \texttt{Nat}$. But allowing this in general would cause severe problems. To see where the problem lies, consider the type $\texttt{Bad}$ with just one constructor

$$\texttt{bad} : (\texttt{Bad} \to \bot) \to \texttt{Bad}$$

We claim that this definition leads to a contradiction. To show this, we will define a function $f : \texttt{Bad} \to \bot$. If we can do this, then we have $\texttt{bad}\ f : \texttt{Bad}$ and hence $f\ (\texttt{bad}\ f) : \bot$. So just by defining the type $\texttt{Bad}$, we have introduced a closed term of type $\bot$, which is supposed to be empty. The logical interpretation of the type $\texttt{Bad}$ makes the problem especially clear: it says "The proposition $\texttt{Bad}$ is true if and only if $\texttt{Bad}$ is false." The 'if' part holds because of the type of the constructor $\texttt{bad}$, while the 'only if' part holds because $\texttt{Bad}$ is the *smallest* type that contains all terms $\texttt{bad}\ x$. It is no wonder that definitions like this lead to inconsistencies.

Here is the definition of the function f:

$$f\ (\texttt{bad}\ x) = x\ (\texttt{bad}\ x)$$

This definition uses pattern matching, which we haven't formally defined yet, but we hope the idea is clear. Remark that

$$f\ (\texttt{bad}\ f) \longrightarrow f\ (\texttt{bad}\ f)$$

so evaluation of this term will never terminate, i.e. $f\ (\texttt{bad}\ f)$ does not have a normal form.

To avoid the definition of problematic types like $\texttt{Bad}$, we will put a condition on the types of the arguments of constructors. If $c : \Phi \to D\ \bar{\imath}$ is a constructor of D, then D may only occur *strictly positively* in each type of Φ. We say that D is strictly positive in a

type T if it only occurs to the right of every arrow $\to$ in T. This rules out constructors like $\texttt{bad} : (\texttt{Bad} \to \bot) \to \texttt{Bad}$ because Bad occurs to the left of an arrow in the argument type $(\texttt{Bad} \to \bot)$.

Definition 2.9 (Strictly positive occurrence). *A (type) variable X is* strictly positive *in the type T if X does not occur in T or if T is of the form $\Psi \to X\ \bar{u}$ where X does not occur in Ψ.*

2.4.3 Definition

Now we can give the formal definition of inductive families.

Definition 2.10 (Inductive family). *Let Γ, Δ, $\Phi_1, \ldots, \Phi_n$ be contexts and $\bar{\imath}_1, \ldots, \bar{\imath}_n$ be lists of terms such that $\Gamma\Delta$ **valid** and $\Gamma(D : \Delta \to Set_l)\Phi_k \vdash \bar{\imath}_k : \Delta$ for $k = 1, \ldots, n$. Suppose D only occurs strictly positively in $\Phi_1, \ldots, \Phi_n$. Then we can define an* inductive family *with* constructors $c_1, \ldots, c_n$ *by the following rules:*

$$\frac{\Gamma' \ \textbf{\textit{valid}}}{\Gamma'\Gamma \vdash D : \Delta \to Set_l}$$

$$\frac{\Gamma' \ \textbf{\textit{valid}}}{\Gamma'\Gamma \vdash c_k : \Phi_k \to D\ \bar{\imath}_k}$$

Constructors tell us how to construct terms of an inductive family, but not how to use them. Classically, the definition of each inductive family is accompanied by an *induction principle* that describes how terms of that type can be used. While it is possible to write programs and proofs using these induction principles, it is very tedious to do so in real applications. The alternative is to extend type theory with *pattern matching*, which is what we will do in the next chapter.

2.5 Fundamental theorems

In this section, we will describe some of the most important theorems of type theory. These theorems are important for several reasons. One reason is that type theory can also be used as a logic, so we cannot allow e.g. infinitely long proofs. Another reason is that these results are needed to keep type checking decidable in the presence of dependent types. Most of the theorems given here are proven in [Gog94], which gives a thorough description of the metatheory of UTT.

The first theorem tells us that each term can have at most one type.

Theorem 2.11 (Unique types). *If $\Gamma \vdash t : T_1$ and $\Gamma \vdash t : T_2$ then $\Gamma \vdash T_1 = T_2 : Set_i$.*

Proof. Corollary 6.8.1 of [Gog94]. □

Using the fact that each term has a unique type and the typing rules of the previous chapter, we can use the form of the term to learn something about the form of its type:

- If a term of the form $\lambda(x : A).\ t$ has any type, then it is of the form $(x : A) \to B$.

- If a term of the form $(x : A) \to B$ has any type, then it is Set_i for some i.
- Set_i always has type Set_{i+1}.

The second theorem tells us that the type of a term is preserved under evaluation. This is essential because $t \longrightarrow t'$ implies that t and t' represent the same object, and that t' is just a 'simpler' representation of this object than t.

Lemma 2.12 (Subject Reduction). *If $\Gamma \vdash t : T$ and $t \longrightarrow t'$ then $\Gamma \vdash t' : T$*

Proof. Corollary 6.8.5 of [Gog94]. □

The next theorem tells us that there can be no infinite evaluation sequences. For type theory as a programming language, this means that well-typed programs will always halt. For type theory as a logic, this means that there can be no infinitely long proofs.

Definition 2.13 (Strong normalization). *An evaluation rule $\longrightarrow$ is* strongly normalizing *if there is no infinite sequence of terms such that $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$*

Theorem 2.14. *The $\beta\eta$-reduction rule $\longrightarrow$ is strongly normalizing.*

Proof. Corollary 6.8.4 of [Gog94]. □

Theorem 2.16 assures us that no matter which evaluation rules we apply first, we will always get the same result. This allows us to speak about *the* normal form of a term.

Definition 2.15 (Church-Rosser property). *An evaluation rule $\longrightarrow$ has the* Church-Rosser property *if the following holds: if for any term t we have $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$ then there exists a term t' such that $t_1 \longrightarrow^* t'$ and $t_2 \longrightarrow^* t'$.*

Theorem 2.16 (Church-Rosser property). *The $\beta\eta$-reduction $\longrightarrow$ has the Church-Rosser property.*

Proof. Corollary 6.8.6 of [Gog94]. □

Corollary 2.17 (Unicity of normal forms). *Two terms are definitionally equal if and only if they have the same normal form.*

This corollary implies that definitional equality of two terms of a given type is decidable.

As mentioned at the start of section 2.4, terms of an inductive family can be constructed by using its constructors, and *only* by using its constructors. We will now turn this into a precise statement.

Definition 2.18 (Constructor form). *A term t is in* constructor form *if it is of the form $c\ t_1\ \ldots\ t_n$ for a constructor c.*

Definition 2.19 (Completeness). *An evaluation rule $\longrightarrow$ is* complete *if the following holds: for any inductive family D and any closed term $t : D\ \bar{\imath}$ in normal form we have that t is in constructor form.*

Lemma 2.20. *If t is a closed normal form of type $(x_1 : A_1) \dots (x_n : A_n) \to D\ i_1\ \dots\ i_n$ then t is either of the form $\lambda(x : A).u$ or $c\ t_1\ \dots\ t_m$ where c is a constructor of the family D.*

Proof. By theorem 2.11 (unicity of types), t is either a variable, a lambda abstraction $\lambda(x : A).\ u$, an application $u\ v$, or a constructor c.

- Because t is a closed term, it cannot be a variable.
- If t is a lambda abstraction then we have proven the statement.
- If t is an application $u\ v$, then u must also be a normal form. By induction on the size of t, we have that u is either of the form $\lambda(x : A).\ s$ or $c\ u_1\ \dots\ u_m$. But in the first case we have that $t = (\lambda(x : A).\ s)\ v$ is not a normal form, a contradiction. So we must have $t = c\ u_1\ \dots\ u_n\ v$, which is of the correct form.
- If t is a constructor c, then it is also of the correct form. □

Theorem 2.21. *The $\beta\eta$-evaluation rule $\longrightarrow$ is complete.*

Proof. Let $t : D\ \bar{\imath}$ be a closed term in normal form. Lemma 2.20 gives us that t is of the form $\lambda(x : A).\ u$ or $c\ t_1\ \dots\ t_m$. But if t is a lambda abstraction, then it would have a type of the form $(x : A) \to B$, a contradiction. So t is of the form $c\ t_1\ \dots\ t_m$ for a constructor c of the family D, as we wanted to prove.

Combining the above theorem with theorem 2.14, we get that any closed term t in an inductive family evaluates to a constructor form. So constructor forms are the 'canonical forms' of an inductive family. We can interpret this theorem as a progress theorem for type theory: the evaluation of a well-typed term never gets stuck.

Chapter 3

Pattern Matching

Terms in constructor form play a special role in inductive families. Indeed, theorem 2.21 tells us that all closed normal forms of an inductive family are in constructor form. So in order to define a function taking inputs from an inductive family, it is sufficient to define it for all inputs in constructor form. For example, to define a function $f : \texttt{Nat} \to T$, it is sufficient to define $f\ \texttt{zero}$ and $f\ (\texttt{suc}\ n)$ for an arbitrary $n : \texttt{Nat}$. This is the principle on which pattern matching is based.

The kind of pattern matching with dependent types described in this chapter was first described by T. Coquand in [Coq92]. A useful extension of pattern matching (which is not described in this thesis) are 'with'-clauses as described in [MM04]. It has also be implemented in multiple dependently typed languages, such as DML[1] [Xi03], Agda [Nor07], and Coq [Soz10].

3.1 Simple pattern matching

We will start by describing pattern matching where all argument types are simple (i.e. non-dependent) types. The result type can be a dependent type, however. This will allow us to discuss important facets of pattern matching before introducing complications like inaccessible patterns and absurd patterns, which are needed to deal with dependent types.

3.1.1 Patterns and clauses

A pattern is a special kind of term that consists of only constructors and free variables. These free variables in a pattern are called the *pattern variables*. The pattern variables in a pattern are usually required to be distinct, if this condition is satisfied then the pattern is called *linear*. Suppose we have a pattern p and an arbitrary term t, then we say that t *matches* p if there is a substitution σ with as its domain the pattern variables of p such that $t = p\sigma$. This is written as $\textsc{match}(p, t) \Rightarrow \sigma$. If there is no such substitution, we write $\textsc{match}(p, t) \Uparrow$. Figure 3.1 describes an algorithm that determines whether a term t matches a pattern p. This is not hard because patterns have a simple tree-like structure. The algorithm described by these rules is called *pattern matching*. Remark that it is

[1] DML is no longer under development, but the same kind of pattern matching is used in its successor ATS.

$$\frac{}{\textsc{match}(x, t) \Rightarrow [x \mapsto t]}$$

$$\frac{\textsc{match}(\bar{p}, \bar{t}) \Rightarrow \sigma}{\textsc{match}(c\ \bar{p}, c\ \bar{t}) \Rightarrow \sigma} \qquad \frac{\textsc{match}(\bar{p}, \bar{t}) \Uparrow}{\textsc{match}(c\ \bar{p}, c\ \bar{t}) \Uparrow}$$

$$\frac{c_1 \neq c_2}{\textsc{match}(c_1\ \bar{p}, c_2\ \bar{t}) \Uparrow}$$

$$\frac{}{\textsc{match}(\epsilon, \epsilon) \Rightarrow []}$$

$$\frac{\textsc{match}(p, t) \Rightarrow \sigma \qquad \textsc{match}(\bar{p}, \bar{t}) \Rightarrow \sigma'}{\textsc{match}(p\ \bar{p}, t\ \bar{t}) \Rightarrow \sigma; \sigma'}$$

$$\frac{\textsc{match}(p, t) \Uparrow}{\textsc{match}(p\ \bar{p}, t\ \bar{t}) \Uparrow} \qquad \frac{\textsc{match}(\bar{p}, \bar{t}) \Uparrow}{\textsc{match}(p\ \bar{p}, t\ \bar{t}) \Uparrow}$$

Figure 3.1: Rules describing the pattern matching algorithm.

essential that the patterns are linear for this algorithm to work, otherwise the variables in the domain of the substitution would not be disjoint.

To define a new function $f : (a : A) \to B\ a$, we give a set of equations of the form $f\ p = t$ where p is a pattern of type A and t is a term of type $B\ p$ that can contain the pattern variables of p. Such an equation is called a *clause* of the definition, and the term t is called the *right-hand side* of the clause. Here is an example of a definition by pattern matching of the function $\texttt{pred} : \texttt{Nat} \to \texttt{Nat}$ that calculates the predecessor:

$$\begin{array}{lcl} \texttt{pred}\ \ \texttt{zero} & = & \texttt{zero} \\ \texttt{pred}\ \ (\texttt{suc}\ n) & = & n \end{array} \tag{3.1}$$

It is obvious how a definition by pattern matching works: to evaluate $f\ s$, we search for a clause $f\ p = t$ such that $\textsc{match}(p, s) \Rightarrow \sigma$. If we find such a clause, then we have $f\ s = f\ (p\sigma) \longrightarrow t\sigma$.

It is allowed that the right-hand side of a clause contains recursive calls to the function we are defining. For example,

$$\begin{array}{lcl} \texttt{double}\ \ \texttt{zero} & = & \texttt{zero} \\ \texttt{double}\ \ (\texttt{suc}\ n) & = & \texttt{suc}\ (\texttt{suc}\ (\texttt{double}\ n)) \end{array} \tag{3.2}$$

We can also define functions $f : (a_1 : A_1) \ldots (a_n : A_n) \to B\ a_1\ \ldots\ a_n$ with multiple arguments by working with lists of patterns. Each clause of f is of the form $f\ p_1\ \ldots\ p_n = t$. The linearity condition now implies that the pattern variables of all the patterns $p_1, \ldots, p_n$ should be different. For example,

$$\begin{array}{lcl} \texttt{plus}\ \ \texttt{zero}\ \ n & = & n \\ \texttt{plus}\ \ (\texttt{suc}\ m)\ \ n & = & \texttt{suc}\ (\texttt{plus}\ m\ n) \end{array} \tag{3.3}$$

3.1.2 Required properties

If we allow any definition by pattern matching, then we encounter a number of problems:

- It might be that there is a closed normal form s that matches none of the patterns in the definition of f. Then $f\ s$ doesn't evaluate any further, hence it is a normal form. Hence it is possible to break completeness theorem 2.21.
- Because the right-hand side of a clause can contain recursive calls to the function f, it is possible that the evaluation of the function f never stops. For example, there can be a clause of the form $f\ x = f\ (\texttt{suc}\ x)$. Hence it is possible to break the strong normalization theorem 2.14.
- It is also possible that a term matches multiple patterns with unequal right-hand sides. Hence it is possible to break the Church-Rosser property 2.16.

To avoid these problems, we will give a number of conditions that definitions by pattern matching must satisfy. Specifically, we require the following three conditions, corresponding to the three above problems:

Completeness For each closed term s, there must be a pattern p in the definition such that s matches p.

Termination There can be no infinite sequence of evaluation steps $f\ a \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$ where f occurs in each of the t_i.

Confluence If there are two clauses $f\ p_1 = t_1$ and $f\ p_2 = t_2$ and substitutions σ_1, σ_2 such that $p_1\sigma_1 = p_2\sigma_2$, then $t_1\sigma_1$ and $t_2\sigma_2$ must be equal.

When these three requirements are satisfied, then we can safely add the new constant f to the theory, together with the evaluation rules given by the clauses of f.

3.1.3 First-match semantics

While the requirements for completeness and termination are conventional, our requirement that definitions must be confluent is not. Instead, overlapping patterns are usually interpreted on a first-match basis: only the first matching clause is considered. This allows us for example to give a definition of $\texttt{equal} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}$ as follows:

$$\begin{array}{llll} \texttt{equal} & \texttt{zero} & \texttt{zero} & = \texttt{true} \\ \texttt{equal} & (\texttt{suc}\ m) & (\texttt{suc}\ n) & = \texttt{equal}\ m\ n \\ \texttt{equal} & m & n & = \texttt{false} \end{array} \tag{3.4}$$

Clearly, the last clause does not hold as a definitional equality but rather only holds when the first two clauses don't match.

If we use the first-match semantics for overlapping patterns, it is always possible to give an equivalent definition without overlapping patterns. In our example, this can be done by replacing the last clause by these two clauses:

$$\begin{array}{llll} \texttt{equal} & \texttt{zero} & (\texttt{suc}\ n) & = \texttt{false} \\ \texttt{equal} & (\texttt{suc}\ m) & \texttt{zero} & = \texttt{false} \end{array} \tag{3.5}$$

In this case there is only one extra clause in the expansion, but this number can increase sharply with the number of constructors. So the main advantage of the first-match semantics is that it allows us to give shorter definitions.

In section 4.4, we will see how the first-match semantics can be used in the translation from a set of clauses to a case tree. In chapter 5, we will discuss a number of problems that are caused by this translation. This will lead to a different approach where we treat clauses with overlapping patterns as definitional equalities, hence the need for the confluence requirement.

3.2 Inaccessible patterns

When we want to generalize pattern matching to functions with dependent types in the argument position, some new complexities arise. For example, it will not be sufficient to use patterns built of only constructors and variables. Inaccessible patterns will fill this gap, by allowing us to exploit the full power of inductive families.

Let's look at the following example: suppose we want to define a function $\texttt{root} : (n : \texttt{Nat})(p : \texttt{Square}\ n) \to \texttt{Nat}$ that, given a natural number n and a proof p that n is a perfect square, computes the square root of n. Remark that by definition of the type $\texttt{Square}\ n$, any canonical term of this type must be of the form $\texttt{sq}\ m$ for some m with $m^2 = n$. But this is exactly the square root we are looking for! So we can simply pattern match on the proof p to get the square root of n. The definition of `root` consists of the single clause $\texttt{root}\ n\ (\texttt{sq}\ m) = m$. There is only one problem with this clause: it is not well-typed. The pattern $\texttt{sq}\ m$ has type $\texttt{Square}\ m^2$, while a pattern of type $\texttt{Square}\ n$ is expected. These types are only equal if $n = m^2$, so we might try to change the clause to $\texttt{root}\ m^2\ (\texttt{sq}\ m) = m$. But this gives two new problems: the pattern is not linear because the variable m occurs twice, and m^2 is neither a constructor nor a variable.

To deal with situations where there is *only one* type-correct term for a pattern variable, we introduce a new type of patterns, called *inaccessible patterns*. We write inaccessible patterns as $\lfloor t \rfloor$ where t is an arbitrary term. For example, we can now write the definition of `root` as $\texttt{root}\ \lfloor m^2 \rfloor\ (\texttt{sq}\ m) = m$. The intuitive meaning of an inaccessible pattern $\lfloor t \rfloor$ is that t is the *only* type-correct term at this position in the pattern.

So in general, a pattern is built up from pattern variables, constructors, and inaccessible patterns. The linearity condition now requires that each pattern variable occurs only once *in an accessible position*. However, there is no limit on how many times it can occur in the inaccessible patterns.

When doing pattern matching, inaccessible patterns seem to pose a significant problem because we have to match with an arbitrary term (not just a pattern). But this is not a real problem: because inaccessible patterns are only allowed when there is only one possibility, they are guaranteed to match whenever the other parts of the patterns do. For example, when calling the function `root` with arguments n and $\texttt{sq}\ m$ we don't have to check whether n is equal to m^2; it is guaranteed by the type system. Hence, we can easily extend the algorithm for pattern matching by the rule given in figure 3.2.

$$\frac{}{\textsc{match}(\lfloor s \rfloor, t) \Rightarrow []}$$

Figure 3.2: Extending the pattern matching algorithm with inaccessible patterns.

3.3 Formal description

We will now give a formal description of definitions by pattern matching and the requirements they have to satisfy. We give only the abstract definitions in this section; how these requirements can actually be checked by a computer is discussed in the next chapter.

3.3.1 Definitions by pattern matching

Definition 3.1 (Pattern). *Let Δ be a valid context.* Patterns *(with pattern variables Δ) are inductively defined as follows:*

- *If $(x : A) \in \Delta$, then x is a pattern.*
- *If c is a constructor of arity n and $p_1, \ldots, p_n$ are patterns, then $c\ p_1\ \ldots\ p_n$ is a pattern.*
- *If $\Delta \vdash t : T$ (i.e. all free variables of t come from Δ), then $\lfloor t \rfloor$ is a pattern.*

Definition 3.2 (Linear pattern). *A list of patterns $\bar{p}$ with pattern variables Δ is* linear *if each pattern variable in Δ occurs exactly once in an accessible position in $\bar{p}$.*

Although patterns look like terms, they form a distinct syntactic class. Definition 3.3 allows us to convert patterns back to terms.

Definition 3.3 (Underlying term of a pattern). *The* underlying term $\lceil p \rceil$ *of a pattern p is defined as follows:*

- $\lceil x \rceil = x$
- $\lceil c\ p_1\ \ldots\ p_n \rceil = c\ \lceil p_1 \rceil\ \ldots\ \lceil p_n \rceil$
- $\lceil \lfloor t \rfloor \rceil = t$

The underlying term $\lceil p_1\ \ldots\ p_n \rceil$ of a list of patterns is the list of underlying terms $\lceil p_1 \rceil\ \ldots\ \lceil p_n \rceil$.

If p is a pattern with pattern variables Δ and $\Gamma\Delta \vdash \lceil p \rceil : T$, then we say that p is a pattern of type T and we write this judgment as $\Gamma|\Delta \vdash p : T$ **pattern**. The vertical bar $|$ separates the regular variables from the pattern variables. Similarly we write $\Gamma|\Delta \vdash \bar{p} : \Theta$ **pattern** if $\Gamma\Delta \vdash \lceil \bar{p} \rceil : \Theta$. We say that the patterns p_1 and p_2 are *equal as terms* if their underlying terms are equal.

Definition 3.4 (Pattern matching). *A list of terms* $\Gamma \vdash \bar{t} : \Theta$ matches *a list of patterns* $\Gamma|\Delta \vdash \bar{p} : \Theta$ ***pattern*** *if there exists a substitution* σ *with domain* Δ *such that* $\lceil \bar{p} \rceil \sigma = \bar{t}$.

Definition 3.5 (Clause). *The possible* clauses *for a function* $f : \Phi \to T$ *are defined by the following inference rule:*

$$\frac{\Gamma|\Delta \vdash \bar{p} : \Phi \ \textbf{\textit{pattern}} \qquad \Gamma(f : \Phi \to T)\Delta \vdash t : T[\Phi \mapsto \lceil \bar{p} \rceil]}{\Gamma \vdash f\ \bar{p} = t \ \textbf{\textit{clause}}}$$

Definition 3.6. *Let* $\longrightarrow$ *be an evaluation rule and* C *a set of clauses for* f. *We define* $\longrightarrow_C$ *as the smallest evaluation rule extending* $\longrightarrow$ *that satisfies* $f\ (\bar{p}\sigma) \longrightarrow_C t\sigma$ *for each clause* $f\ \bar{p} = t$ *in* C *and each substitution* σ *with domain the pattern variables of* $\bar{p}$.

3.3.2 Conditions on the inaccessible patterns

We have said that inaccessible patterns can only be used when there is only one possible type-correct term at that position. But this is not a very precise definition. In this section, we will define the notion of a *valid pattern* that gives a precise meaning to this intuition. First, we need two auxiliary definitions:

Definition 3.7 (Skeleton). *Let* ω *be a fixed symbol which is not a variable or a constructor. The* skeleton *of a pattern is defined as follows:*

- *The skeleton of a pattern variable* x *or an inaccessible pattern* $\lfloor t \rfloor$ *is always* ω.
- *The skeleton of a constructor pattern* $c\ p_1\ \ldots\ p_n$ *is* $c\ s_1\ \ldots\ s_n$, *where* $s_1, \ldots, s_n$ *are the skeletons of* $p_1, \ldots, p_n$.

In other words, the skeleton of a pattern remembers the constructors, but forgets all variables and inaccessible patterns.

Definition 3.8 (Pattern specialization). *Let* $\bar{p}, \bar{q}$ *be patterns. We say that* $\bar{p} \supseteq \bar{q}$ *(*$\bar{q}$ *is a* specialization *of* $\bar{p}$*) if there exists a substitution* δ *such that* $\lceil \bar{q} \rceil = \lceil \bar{p} \rceil \delta$ *and all variables in the domain of* δ *are pattern variables of* $\bar{p}$.

It is easy to see that if $\bar{p} \supseteq \bar{q}$ and a list of terms $\bar{t}$ matches $\bar{q}$, then $\bar{t}$ also matches $\bar{p}$. This implies that the set of lists of terms matching $\bar{q}$ is a subset of the lists of terms matching $\bar{p}$, hence the notation.

Intuitively, a pattern is valid if it is the most general pattern with the same skeleton. This idea is captured in definition 3.9.

Definition 3.9 (Valid pattern). *A list of patterns* $\Gamma|\Delta \vdash \bar{p} : \Phi$ ***pattern*** *is* valid *if for each other list of patterns* $\Gamma|\Delta' \vdash \bar{p}' : \Phi$ ***pattern*** *with the same skeleton we have* $\bar{p} \supseteq \bar{p}'$.

How does this definition prevent 'bad' inaccessible patterns? Suppose a pattern p contains an inaccessible pattern $\lfloor t \rfloor$ but there is another type-correct term t' for this

position with $t' \neq t$. Then we can also form a pattern p' that is equal to p except that $\lfloor t \rfloor$ has been replaced by $\lfloor t' \rfloor$. Then p and p' have the same skeleton. However, there is no substitution σ with in its domain only pattern variables of p such that $\lceil p' \rceil = \lceil p \rceil \sigma$. Hence p is not valid.

Valid patterns are useful because they guarantee us that inaccessible patterns will always match whenever the accessible parts of the pattern match. This is made precise in lemma 3.10.

Lemma 3.10 (Respectful patterns). *Suppose* $\Gamma | \Delta \vdash \bar{p} : \Phi$ ***pattern****. Then* $\bar{p}$ *is a valid list of patterns if and only if* $\textsc{match}(\bar{p}, \bar{t}) \Rightarrow \sigma$ *implies that* $\lceil \bar{p} \rceil \sigma = \bar{t}$ *(i.e.* $\bar{t}$ *matches* $\bar{p}$*).*

Proof. Suppose $\bar{p}$ is valid and $\textsc{match}(\bar{p}, \bar{t}) \Rightarrow \sigma$. Then $\bar{t}$ matches all accessible parts of $\bar{p}$. This means that there exists a pattern $\bar{p}'$ with the same skeleton as $\bar{p}$ such that $\lceil \bar{p}' \rceil = \bar{t}$. Remark that $\bar{p}'$ has no pattern variables, only constructors and inaccessible patterns. Because $\bar{p}$ is a valid pattern, there exists a substitution σ' such that $\lceil \bar{p} \rceil \sigma' = \lceil \bar{p}' \rceil = \bar{t}$. By definition of MATCH, we must have that σ and σ' are equal on all pattern variables of $\bar{p}$. But the domain of both σ and σ' is exactly the set of pattern variables of $\bar{p}$ by definition, hence $\sigma = \sigma'$. We can conclude that $\lceil \bar{p} \rceil \sigma = \bar{t}$.

Conversely, suppose $\textsc{match}(\bar{p}, \bar{t}) \Rightarrow \sigma$ implies that $\lceil \bar{p} \rceil \sigma = \bar{t}$. Let $\bar{p}'$ be any pattern with the same skeleton as $\bar{p}$. Then we have $\textsc{match}(\bar{p}, \lceil \bar{p}' \rceil) \Rightarrow \sigma$ for some substitution σ satisfying $\lceil \bar{p} \rceil \sigma = \lceil \bar{p}' \rceil$. So we can conclude that $\bar{p}$ is a valid pattern. □

3.3.3 Required properties

Just as for simple pattern matching, we still need to assure that definitions by pattern matching don't cause anything bad to happen. Here are the conditions in their complete form:

Definition 3.11 (Completeness). *A set of lists of patterns* $\Gamma | \Delta_i \vdash \bar{p}_i : \Phi$ ***pattern*** *for* $i = 1, \ldots, n$ *is* complete *if for each list of closed terms* $\bar{t} : \Phi$ *there exists an index* i *such that* $\bar{t}$ *matches* $\bar{p}_i$*.*

In other words, if the patterns of a set of clauses C are complete, then the evaluation rule $\longrightarrow_C$ is complete.

Definition 3.12 (Termination). *A set of clauses* C *is* terminating *if the evaluation rule* $\longrightarrow_C$ *is strongly normalizing.*

Definition 3.13 (Confluence). *A set of clauses* C *is* confluent *if the evaluation rule* $\longrightarrow_C$ *has the Church-Rosser property.*

Definition 3.14 (Definition by pattern matching). *Let* C *be a set of clauses for* $f : \Phi \to T$ *in a context* Γ *such that:*

- *All patterns in the clauses of* C *are valid.*
- *The patterns of the clauses in* C *are complete.*

- *The set of clauses C is terminating.*
- *The set of clauses C is confluent.*

Then we can define the function f by adding the inference rule

$$\frac{\Gamma \ \textbf{\textit{valid}}}{\Gamma \vdash f : \Phi \to T}$$

and for each clause $f\ \bar{p} = t$ in C the inference rule

$$\frac{\Gamma \vdash \bar{s} : \Phi \qquad \textsc{match}(\bar{p}, \bar{s}) \Rightarrow \sigma}{\Gamma \vdash f\ \bar{s} = t\sigma : T[\Phi \mapsto \bar{s}]}$$

From now on, we will denote $\longrightarrow$ for the evaluation rule containing all evaluation rules $\longrightarrow_C$ for each function f defined by a set of clauses C. Remark that each clause of the definition holds as a definitional equality, unlike with the first-match semantics.

Theorem 3.15. *The evaluation rule $\longrightarrow$ is complete, strongly normalizing, and Church-Rosser.*

Proof. By construction. □

3.4 Pattern matching versus eliminators

There is a general method to equip each inductive family with an *eliminator*, as we remarked in the previous chapter. This eliminator embodies the induction principle for that particular inductive family. For example, the eliminator for `Nat` is `natrec` and the eliminator for `Bool` is `if · then · else ·`.[2] The general form of the eliminator of an inductive family is described in for example [Luo94] and [GMM06].

While eliminators are certainly sufficient to write all kinds of proofs and programs on inductive families, they are not very convenient. That is why we choose to use pattern matching instead to define those proofs and programs. This allows us to give those definitions in an easier and more readable way.

Remark that each set of patterns defines a new induction principle. This means we have no need for the classic eliminators for inductive families, they can be defined using pattern matching. For example, the eliminator of the identity type $\mathtt{J} : (b : A) \to (p : a \equiv_A b) \to C\ a\ (\mathtt{refl}\ a) \to C\ b\ p$ can be defined by the single clause

$$\mathtt{J}\ \lfloor a \rfloor\ (\mathtt{refl}\ \lfloor a \rfloor)\ c = c \tag{3.6}$$

Be aware though that while pattern matching is more convenient than using the induction principles, it is also harder to understand theoretically. It is a deep and nontrivial result that under certain constraints, pattern matching can be written in function of the induction principles plus the so-called K axiom. This elimination of definitions by pattern matching is described in [McB00] and [GMM06].

[2] In fact, the real eliminators have a more general type than the examples given here.

Chapter 4

Checking pattern matching definitions

In the previous chapter, we described dependent pattern matching and how we can use it to define (dependent) functions. We also gave a number of conditions that these definitions have to satisfy in order to be well-defined: pattern validity, completeness, termination, and confluence. However, the conditions were formulated in an abstract way that is not easily checked by a computer. In order to use pattern matching in implementations of type theory such as Agda, we need more concrete criteria that the definitions have to satisfy. In this chapter, we will describe the algorithms used in Agda as described in [Nor07]. We will also describe the representation of pattern matching definitions by case trees, which are used to evaluate functions efficiently. The nice thing is that we get this representation 'for free' when we check completeness.

4.1 Checking inaccessible patterns

Inaccessible patterns are introduced by constraints on the types of the constructors in the pattern. For example, the inaccessible pattern $\lfloor m^2 \rfloor$ in the definition of `root` (given in section 3.2) was introduced because the type of `sq` m is `Square` m^2. So when type checking the definition of `root`, we are faced with the following problem: under what conditions can use a pattern of the form $c\ x_1\ \ldots\ x_n$ of type $D\ \bar{\imath}$ and which constraints do the indices $\bar{\imath}$ have to satisfy?

If T is a simple inductive type like `Nat` or `Bool` without indices, then the answer is easy: any constructor c of the type T can be used, and no extra constraints are needed. If T is part of an inductive family, the situation is more complex. For example, suppose $T = x \leq y$ for concrete values of x, y : `Nat` (see section 2.4.1 for the definition of the type $x \leq y$). There are now three possibilities for the set of constructors of T:

- If x is strictly smaller than y, then the only valid constructor of $x \leq y$ is `smaller` : $(n$: `Nat`$)(m$: `Nat`$)(p : n \leq m) \rightarrow n \leq$ (`suc` m). This gives the constraints that $x = n$ and $y =$ `suc` m.

- If x and y are equal, then the only valid constructor of $x \leq y$ is `equal` : $(n$: `Nat`$) \rightarrow n \leq n$. This gives the constraint that $x = y = n$.

- If x is bigger than y, then there are no valid constructors and hence $x \leq y$ is empty.

This problem can be solved in general by *unifying* the indices of T with the indices of the type of each constructor.

4.1.1 Unification

Suppose we want to unify two terms a and b, i.e. we are interested in substitutions δ such that $a\delta = b\delta$. Such a substitution is called a *unifier* of a and b. A *most general unifier* of a and b is a unifier δ such that for each other unifier δ', there exists a substitution σ such that $\delta' = \delta; \sigma$.

The question whether unifiers exist is called the *unification problem*. In general, this is an undecidable problem. There exist unification algorithms that search for a most general unifier (we will give one below) but they can give up in case the unification problem is too hard. We say that a unification algorithm succeeds positively if it finds a most general unifier, that it succeeds negatively if it concludes that there exist no unifiers, and that it fails if it gives up.

Returning to the previous example, to know whether `smaller` is a constructor of the type $x \leq y$, we have to unify the types $n \leq (\texttt{suc}\ m)$ and $x \leq y$. The unification algorithm will succeed positively with result $[x \mapsto n, y \mapsto \texttt{suc}\ m]$.

Remark that a most general unifier of two terms is not necessarily unique: in the example, $[n \mapsto x, y \mapsto \texttt{suc}\ m]$ is also a most general unifier. In a situation like this, the unification algorithm has to make an arbitrary choice between the two unifiers. To avoid this problem, we will supply the algorithm with a set of variables called the *flexible variables*. The flexible variables are the variables that can be used for the unification, i.e. that can occur in the domain of the substitution. If we let x and y be the flexible variables in the example, then the unification algorithm will always return the substitution $[x \mapsto n, y \mapsto \texttt{suc}\ m]$.

The unification algorithm is described in figure 4.1. This algorithm is the same as the one used in [Nor07]. The algorithm has as input two terms a and b and a set of flexible variables ξ. We write $\textsc{unify}(u, v) \Rightarrow \sigma$ if unification of u and v succeeds positively and $\textsc{unify}(u, v) \Uparrow$ if unification succeeds negatively. If we have neither $\textsc{unify}(u, v) \Rightarrow \sigma$ nor $\textsc{unify}(u, v) \Uparrow$, then the algorithm fails and we write $\textsc{unify}(u, v) \stackrel{?}{\Rightarrow}$.

4.1.2 Specialization steps

Now we know how to do unification, we can use it to build valid patterns.

Definition 4.1 (Specialization step). *Suppose* $\Gamma|\Delta \vdash \bar{p} : \Phi$ ***pattern*** *and let* $(x : D\ \bar{\imath}) \in \Delta$ *be a pattern variable of type* $D\ \bar{\imath}$ *where* D *is an inductive family. Let* $c : \Theta \to D\ \bar{\jmath}$ *be a constructor of* D*.*

- *If* $\textsc{unify}(\bar{\imath}, \bar{\jmath}) \Rightarrow \delta$ *with flexible variables* $\Delta\Theta$*, then we write* $\bar{p} \Rightarrow_c^x \bar{p}'$ *where* $\bar{p}' = \bar{p}([x \mapsto c\ \Theta]; \lfloor\delta\rfloor)$*. Here* $\lfloor\delta\rfloor$ *of a substitution* $\delta = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ *is defined as* $\lfloor\delta\rfloor = [x_1 \mapsto \lfloor t_1\rfloor, \ldots, x_n \mapsto \lfloor t_n\rfloor]$*.*
- *If* $\textsc{unify}(\bar{\imath}, \bar{\jmath}) \Uparrow$ *with flexible variables* $\Delta\Theta$*, then we write* $\bar{p} \Uparrow_c^x$*.*

Remark that $\bar{p} \Rightarrow_c^x \bar{p}'$ implies that $\bar{p} \sqsupseteq \bar{p}'$, so specialization steps give us specialized patterns. The reason why we define specialization steps is because we can use them to

$$\frac{x \in \xi \qquad x \notin FV(v)}{\textsc{unify}(x, v) \Rightarrow [x \mapsto v]}$$

$$\frac{x \in \xi \qquad x \in FV(\bar{v})}{\textsc{unify}(x, c\ \bar{v}) \Uparrow}$$

$$\frac{\textsc{unify}(\bar{u}, \bar{v}) \Rightarrow \sigma}{\textsc{unify}(c\ \bar{u}, c\ \bar{v}) \Rightarrow \sigma}$$

$$\frac{\textsc{unify}(\bar{u}, \bar{v}) \Uparrow}{\textsc{unify}(c\ \bar{u}, c\ \bar{v}) \Uparrow}$$

$$\frac{c_1 \neq c_2}{\textsc{unify}(c_1\ \bar{u}, c_2\ \bar{v}) \Uparrow}$$

$$\frac{\Gamma \vdash u = v : T}{\textsc{unify}(u, v) \Rightarrow []}$$

$$\frac{\textsc{unify}(u, v) \Rightarrow \sigma}{\textsc{unify}(v, u) \Rightarrow \sigma}$$

$$\frac{}{\textsc{unify}(\epsilon, \epsilon) \Rightarrow []}$$

$$\frac{\textsc{unify}(u, v) \Rightarrow \sigma \qquad \textsc{unify}(\bar{u}\sigma, \bar{v}\sigma) \Rightarrow \sigma'}{\textsc{unify}(u\ \bar{u}, v\ \bar{v}) \Rightarrow \sigma; \sigma'}$$

$$\frac{\textsc{unify}(u, v) \Uparrow}{\textsc{unify}(u\ \bar{u}, v\ \bar{v}) \Uparrow}$$

$$\frac{\textsc{unify}(u, v) \Rightarrow \sigma \qquad \textsc{unify}(\bar{u}\sigma, \bar{v}\sigma) \Uparrow}{\textsc{unify}(u\ \bar{u}, v\ \bar{v}) \Uparrow}$$

Figure 4.1: The unification algorithm. The second rule states that unification succeeds negatively when a variable occurs cyclically, what would otherwise result in an infinitely large term.

build valid patterns. This is expressed by lemma 4.2.

Lemma 4.2. *If $\bar{p}$ is a valid pattern of type Φ and $\bar{p} \Rightarrow_c^x \bar{p}'$, then $\bar{p}'$ is also a valid pattern of type Φ.*

Proof. This follows directly from the proof of lemma 11 of [GMM06] and lemma 3.10. □

Definition 4.3 (Specialization sequence). *If there is a sequence $\bar{p}_0 \Rightarrow_{c_1}^{x_1} \bar{p}_1 \Rightarrow_{c_2}^{x_2} \ldots \Rightarrow_{c_n}^{x_n} \bar{p}_n$ of length $n \geq 0$, then we write $\bar{p}_0 \Rightarrow^* \bar{p}_n$.*

Corollary 4.4 (Construction of valid patterns). *Let Φ be a valid context. If $\Phi \Rightarrow^* \bar{p}$, then $\bar{p}$ is a valid pattern of type Φ.*

Now we know how to build valid patterns constructor by constructor. It is also possible to *check* whether a given pattern is valid by attempting to reconstruct it in this way. This is described in sections 2.1.6 and 2.1.7 of [Nor07].

4.2 Checking completeness with coverings

In order to check completeness, we have to make sure that at each position in the pattern, all possible constructors are considered. For example, to define a function $f : \texttt{Nat} \to T$, there should be two clauses $f\ \texttt{zero} = \ldots$ and $f\ (\texttt{suc}\ n) = \ldots$ corresponding to the two constructors `zero` and `suc`. We say that the trivial pattern $m : \texttt{Nat}$ *splits* on m into the two patterns `zero` and `suc` n. We can split the pattern `suc` n again on n to get the two patterns `suc zero` and `suc` (`suc` k). By repeatedly splitting patterns on a pattern variable, we get a *covering*. For example, we just showed that

$$\{\texttt{zero}, \texttt{suc zero}, \texttt{suc}\ (\texttt{suc}\ k)\}$$

is a covering. Since each constructor is considered each time we split a pattern, a covering will always be complete. We will prove this in theorem 4.8.

As in the previous section, we need to be careful if the type of a pattern variable is part of an inductive family. Luckily, we can reuse the unification algorithm to check what constructors can be used. This leads to the definition 4.5.

Definition 4.5 (Splitting/Direct covering). *Suppose $\Gamma|\Delta \vdash \bar{p} : \Phi$ **pattern** and $x : T \in \Delta$ where T is a type from an inductive family with constructors $c_1, \ldots, c_n$. Suppose furthermore that $\bar{p} \Rightarrow_{c_i}^x \bar{p}_i$ or $\bar{p} \Uparrow_{c_i}^x$ for all $i = 1, \ldots, n$. Then we call the set $\{\bar{p}_i \mid \bar{p} \Rightarrow_{c_i}^x \bar{p}_i\}$ a* splitting *or a* direct covering *of $\bar{p}$.*

Lemma 4.6 tells us that splitting a pattern doesn't change which closed terms match the patterns. This is the essential step in showing that a covering is complete.

Lemma 4.6. *If $\{\bar{p}_1, \ldots, \bar{p}_n\}$ is a direct covering of $\Gamma|\Delta \vdash \bar{p} : \Phi$ **pattern** and $\bar{t} : \Phi$ is a closed term matching $\bar{p}$, then $\bar{t}$ matches one of the $\bar{p}_i$.*

Proof. Suppose $\bar{t} : \Phi$ is closed and $\lceil \bar{p} \rceil \sigma = \bar{t}$ for some substitution σ with domain Δ. By definition of a direct covering, there is a variable $(x : D\ \bar{u}) \in \Delta$ and constructors $c_1, \ldots, c_n$

such that $\bar{p} \Rightarrow^x_{c_i} \bar{p}_i$ for $i = 1, \ldots, n$. Let $s = x\sigma$, then s is also a closed term because $\bar{t}$ is. By strong normalization and completeness, we have that $s \longrightarrow^* c\ s_1\ \ldots\ s_k$ for some constructor c of the inductive family D. Let $c : \Psi \to D\ \bar{v}$ be the type of c and $\tau = [\Psi \mapsto s_1\ \ldots\ s_k]$. Remark that we can decompose σ as $[x \mapsto c\ \Psi]; \tau; \sigma'$ where x is not in the domain of σ'.

By type-correctness of $\bar{t} = \lceil \bar{p} \rceil [x \mapsto c\ \Psi] \tau \sigma' : \Phi$, we must have that $\tau; \sigma'$ unifies the types $D\ \bar{u}$ and $D\ \bar{v}$ of x and $c\ \Psi$ respectively. This implies that the unification of $\bar{u}$ and $\bar{v}$ does not succeed negatively, and it also does not fail by definition of a direct covering. So we must have that the unification succeeds positively, i.e. $\bar{p} \Rightarrow^x_c \bar{p}'$ for $\bar{p}' = \bar{p}[x \mapsto c\ \Psi] \lfloor \delta \rfloor$ where δ is a most general unifier of $\bar{u}$ and $\bar{v}$. But $\tau; \sigma'$ was also a unifier of $\bar{u}$ and $\bar{v}$, so there must exist some ρ such that $\tau; \sigma' = \delta; \rho$. Then we have $\bar{t} = \lceil \bar{p} \rceil [x \mapsto c\ \Psi] \tau \sigma' = \lceil \bar{p} \rceil [x \mapsto c\ \Psi] \delta \rho = \lceil \bar{p}' \rceil \rho$.

Again by definition of a direct covering, $\bar{p}'$ must equal $\bar{p}_i$ and c must equal c_i for some i between 1 and n. So we have $\bar{t} = \lceil \bar{p}_i \rceil \rho$, as we wanted to prove. □

Definition 4.7 (Covering). *The* coverings *of $\bar{p}$ are defined as follows:*

1. *The set $\{\bar{p}\}$ is a covering of $\bar{p}$.*
2. *If $\{\bar{p}_1, \ldots, \bar{p}_n\}$ is a direct covering of $\bar{p}$ and $\mathcal{O}_i$ is a covering of $\bar{p}_i$ for $i = 1, \ldots, n$, then $\bigcup_{i=1}^n \mathcal{O}_i$ is a covering of $\bar{p}$.*

A covering *is a covering of the trivial pattern (consisting of only pattern variables).*

Theorem 4.8 follows directly from lemma 4.6.

Theorem 4.8. *A covering is always complete.*

We know that a set of patterns is complete if it forms a covering, but how do we recognize coverings? In general, this problem is known to be undecidable [GMM06]. The problem is the following: an empty covering can be built by an arbitrarily large number of splittings, if all branches lead to a negative unification. So when reconstructing a covering, we cannot know when to stop splitting. The answer to this problem used in [GMM06] and [Nor07] is to require explicit refutations in the pattern if $\bar{p} \Uparrow^x_c$ for all applicable constructors c. So we will extend the syntax of our patterns by the *absurd pattern* $\emptyset$ as follows:

$$\frac{\forall c :\ \bar{p} \Uparrow^x_c}{\bar{p} \Rightarrow^x_\emptyset \bar{p}[x \mapsto \emptyset]}$$

A clause with an absurd pattern is only used to make the job of the completeness checker easier; it will never match a closed term and hence it does not need a right-hand side.

$$\frac{\Gamma | \Delta \vdash \bar{p} : \Phi\ \mathbf{pattern} \qquad \bar{p} \Rightarrow^x_\emptyset \bar{p}'}{\Gamma \vdash f\ \bar{p}'\ \mathbf{clause}}$$

Using these absurd patterns, it again becomes decidable whether a given set of patterns forms a covering [GMM06].

4.3 Checking termination with the structural order

In order to check termination, we have to put a condition on the arguments of the recursive calls in the right-hand side. This should prevent clauses such as

$$f\ (\texttt{suc}\ n) = f\ (\texttt{suc}\ (\texttt{suc}\ n))$$

which lead to non-termination. To do this, it is required that the arguments of the right-hand side recursive calls are built from a strictly smaller number of constructors than the left-hand side pattern. If this is the case, then we call the recursive arguments *structurally smaller*. For example, n is structurally smaller than $\texttt{suc}\ n$, but $\texttt{suc}\ (\texttt{suc}\ n)$ is not. Hence the clause $f\ (\texttt{suc}\ n) = f\ n$ is allowed, but $f\ (\texttt{suc}\ n) = f\ (\texttt{suc}\ (\texttt{suc}\ n))$ is not.

It is possible that a constructor of an inductive type T has an argument of a type of the form $\Delta \to T$ where Δ is a nonempty context. The strictly positiveness requirement ensures us that T does not occur in Δ. For example, we can define a type of infinitely branching trees `InfTree` with two constructors

$$\begin{array}{lcl} \texttt{leaf} & : & \texttt{InfTree} \\ \texttt{branch} & : & (\texttt{Nat} \to \texttt{InfTree}) \to \texttt{InfTree} \end{array}$$

The tree $\texttt{branch}\ h$ has infinitely many direct subtrees, namely one subtree $h\ n$ for each $n : \texttt{Nat}$. While defining a function $f : \texttt{InfTree} \to B$, we have that $h : \texttt{Nat} \to \texttt{InfTree}$ is structurally smaller than $\texttt{branch}\ h$. However, this doesn't allow us to make recursive calls of f on the direct subtrees $h\ n$. To solve this problem, a function h is defined to be structurally as large as its largest value $h\ n$. Formally, this means that if h is structurally smaller than t, then $h\ n$ is also structurally smaller than t for any n. For example, this allows us to define a function $f : \texttt{InfTree} \to \texttt{Nat}$ as follows:

$$\begin{array}{lcl} f\ \ \texttt{leaf} & = & \texttt{zero} \\ f\ \ (\texttt{branch}\ h) & = & \texttt{suc}\ (f\ (h\ \texttt{zero})) \end{array}$$

We are now ready to give the definition of the structural order:

Definition 4.9 (The structural order). *The structural order $\prec$ between terms is inductively defined by the following rules:*

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

If $s \prec t$, then we say that s is structurally smaller *than t.*

This is the same structural order as the one used in [GMM06].

It is nice to know that the arguments of the function become smaller with each recursive call, but this doesn't guarantee termination by itself. Indeed, if there would exist an infinite sequence of terms, each smaller than the last, then evaluation could still go on forever. So we have to check whether there exist any such infinite sequences. If there are no such sequences for a certain order $<$, then the order is called *well-founded*. Theorem 4.10 tells us that this requirement is satisfied for the structural order.

Theorem 4.10. *The structural order $\prec$ is well-founded.*

Proof. A proof of theorem 4.10 is given for a version of λ-calculus with simple types and pattern matching in [AA02]. Since termination is a property of terms and not of their types, one can expect that it is not too hard to adapt the proof to a dependently typed context. In fact, [GMM06] also contains a very indirect proof for the case with dependent types. We will not give the proof here because termination is not the focus of this thesis.□

We are now almost ready to give the criterion for termination. The only case we haven't considered yet, is for functions with multiple arguments. It is not sufficient to require that each clause separately reduces the size of one of the arguments. Consider for example the following two clauses of a function $f : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat}$:

$$\begin{array}{llll} f & \mathtt{zero} & \mathtt{zero} & = \mathtt{zero} \\ f & (\mathtt{suc}\ m) & \mathtt{zero} & = f\ m\ (\mathtt{suc}\ \mathtt{zero}) \\ f & m & (\mathtt{suc}\ n) & = f\ (\mathtt{suc}\ (\mathtt{suc}\ m))\ n \end{array}$$

Then we have

$$f\ (\mathtt{suc}\ m)\ \mathtt{zero} \longrightarrow f\ m\ (\mathtt{suc}\ \mathtt{zero}) \longrightarrow f\ (\mathtt{suc}\ (\mathtt{suc}\ m))\ \mathtt{zero} \longrightarrow \ldots$$

which will go on infinitely. To avoid this problem, it is required that there is a number k such that each clause reduces the size of the k'th argument. This leads to the full criterion for termination given in theorem 4.11.

Corollary 4.11. *Let C be a set of clauses for the function $f : \Delta \to T$ with n patterns each and let $1 \leq k \leq n$. If for each clause $f\ \bar{p} = t$ in C and all recursive calls $f\ \bar{s}$ in t we have that $s_k \prec \lceil p_k \rceil$, then the clauses C are terminating.*

For the proof, we first need the following easy lemma.

Lemma 4.12. *If $s \prec t$, then $s\sigma \prec t\sigma$ for any substitution σ.*

Proof of the lemma. By induction on the definition of $\prec$. □

Proof of corollary 4.11. Let C be a set of clauses defining the function f. Consider the following property $T_f(\bar{s})$ of a list of terms $\bar{s}$ of length n:

> "If $\bar{s}$ has type Δ, then evaluation of $f\ \bar{s}$ always reaches a normal form after a finite number of evaluation steps of $\longrightarrow_C$."

To prove that the clauses C are terminating, we have to prove that $T_f(\bar{s})$ holds for any $\bar{s}$. By the principle of well-founded induction, it is sufficient to prove that $f\ \bar{s}$ reaches a normal form from the assumption that $f\ \bar{s}'$ reaches a normal form for any $\bar{s}' : \Delta$ with $s'_k \prec s_k$. But all recursive calls of f in the right-hand side of the clauses of C satisfy this condition, by assumption of the theorem. By the lemma, this stays true when we apply the function to a concrete argument. Hence we can conclude that the set of clauses C is terminating. □

The current implementation of termination checking in Agda is more complicated than described in this section. For instance, it uses size-change termination [LJB01] and supports corecursive definitions. A good overview of the modern approaches to termination can be found in [VCW12]. [1]

[1] Thanks to Thorsten Altenkirch, Nils Anders Danielsson, and James McKinna for pointing me to the references in this section.

4.4 Case trees

As far as defining functions goes, it is fine to represent a definition by a set of clauses. But if we want to evaluate the function or reason about it, it is better to have a representation that gives more structure. Hence definitions by pattern matching are often represented as *case trees*. A case tree tells us how the patterns of the definition are built by introducing constructors step by step. Each leaf of a case tree for a function f corresponds to a clause for f.

Definition 4.13 (Case tree). *A* case tree *for a function $f : \Delta \to T$ with label $\bar{p} : \Delta$* ***pattern*** *is one of the following:*

- *Either it is an internal node, and the subtrees are again case trees for f such that the labels of the subtrees form a direct covering of $\bar{p}$,*
- *or it is a leaf node containing a term $t : T[\bar{x} \mapsto \lceil \bar{p} \rceil]$ called the right-hand side,*
- *or $\bar{p}$ contains an absurd pattern $\emptyset$, in this case the tree is a leaf node with no right-hand side.*

A case tree for *f is a case tree with as its root label the trivial pattern Δ consisting only of pattern variables.*

Here is an example of a case tree for the function $\texttt{plus} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$:

$$
\begin{array}{l}
n\ m \\
\quad n : \texttt{Nat} \left\{ \begin{array}{l} \texttt{zero}\ m \mapsto m \\ (\texttt{suc}\ n)\ m \mapsto \texttt{suc}\ (\texttt{plus}\ n\ m) \end{array} \right.
\end{array}
$$

There are a few advantages to using case trees instead of plain sets of clauses. Firstly, they give an efficient method to evaluate functions defined by pattern matching. Secondly, the patterns at the leaves of a case tree always form a covering, hence they are complete. Thirdly, each 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].

However, case trees are not sufficient to represent all pattern matching definitions. This problem will become even more apparent when we consider overlapping patterns, since these cannot be represented by case trees at all.

4.4.1 Evaluating a function given by a case tree

If we have a case tree for a function $f : \Delta \to T$, then we can use it to evaluate $f\ \bar{t}$ efficiently without considering all clauses of f separately. This is done by matching the arguments of f against the labels of the case tree. Because the root of a case tree is labeled by the trivial pattern, we know that any arguments will always match the label of the root. We will then recursively look for a subtree such that the arguments match the label of that subtree until we arrive at a leaf node. Finding the correct subtree is done by matching the arguments against the label of each subtree.

If the arguments are *closed* terms, then lemma 4.6 guarantees that at least one subtree will give a match. However, if the arguments contain free variables then it is not always

possible to find a matching subtree. In this case, evaluation gets stuck until the free variable is instantiated.

4.4.2 Building a case tree from a set of clauses

In this section, we will give an algorithm to construct a case tree from a given (complete) set of clauses. This algorithm is adapted from [Nor07]. The algorithm does not require the input patterns to form a covering, so the patterns of the case tree will not be the same as the input patterns. It will however try to build a covering that is as close to the original definition as possible. In particular, when dealing with overlapping patterns, the algorithm will choose whatever pattern comes first. In other words, the resulting case tree follows the first-match semantics of pattern matching.

The algorithm starts with an ordered set of clauses C for a function $f : \Delta \to T$, and its goal is to construct a case tree with label Δ. The algorithm works by recursively constructing a case tree with label $\bar{q}$, while maintaining the invariant that C is nonempty and $\bar{q} \supseteq \bar{p}_i$ for each clause $f\ \bar{p}_i = t_i$ in C, i.e. there exist substitutions δ_i such that $\lceil \bar{p}_i \rceil \lceil \bar{q} \rceil \delta_i$.

If there is a clause $f\ \bar{p}_i = t_i$ in C such that we have $\bar{p}_i \supseteq \bar{q}$, then we have reached a leaf node. The right-hand side is $t_i\sigma$, where σ is the substitution such that $\bar{q} = \bar{p}_i\sigma$. It might be that there are multiple clauses for which $\bar{p}_i \supseteq \bar{q}$, in this case the first clause is chosen.

Otherwise we choose a pattern variable x in $\bar{q}$ such that there is a direct covering $\{\bar{q}_j \mid \bar{q} \Rightarrow^x_{c_j} \bar{q}_j, j = 1, \ldots, k\}$ of $\bar{q}$. For each j from 1 to k, we will recursively build a case tree with label $\bar{q}_j$. To do this, we will divide C into a number of disjoint sets: one set C_j for each constructor c_j and two additional sets C_{var} and C_{rest}.

$$\begin{array}{rcl} C_j & = & \{f\ \bar{p}_i = t_i \mid x\delta_i \text{ is of the form } c_j\ r_1\ \ldots\ r_l\} \\ C_{var} & = & \{f\ \bar{p}_i\ = t_i \mid x\delta_i \text{ is a variable}\} \\ C_{rest} & = & \{f\ \bar{p}_i\ = t_i \mid x\delta_i \text{ is an inaccessible pattern}\} \end{array} \tag{4.1}$$

For each constructor c_j, we also define a set of *specialized* clauses C'_j as follows:

$$C'_j = \{f\ \bar{p}_i\sigma = t_i\sigma \mid \bar{p}_i \Rightarrow^{x\delta_i}_c \bar{p}_i\sigma, (f\ \bar{p}_i = t_i \in C_{var}\} \tag{4.2}$$

The subtree with label $\bar{q}_j$ is now recursively constructed using the clauses $C_j \cup C'_j$. Remark that the original clauses in C_{var} and C_{rest} are thrown away; they are not used in the further construction of the case tree.

Remark that we have not yet specified how to choose the variable x. In order to make progress, we should always choose a *blocking variable*: this is a pattern variable x of $\bar{q}$ such that the corresponding pattern $x\delta_i$ in at least one $\bar{p}_i$ is a constructor pattern. There may be multiple blocking variables; in this situation we have to choose one. The problem is that not all choices give us a complete case tree and even if two different choices both give a case tree, they will not always be equivalent. The consequences of this choice and a possible solution will be discussed in the next chapter.

Chapter 5

Overlapping and Order-independent Patterns

In this chapter, we will present the motivations for the extension of pattern matching discussed in the rest of this thesis. We will first describe the two design goals that make dependent pattern matching as it currently is: the need to write definitions in function of eliminators, and the first-match semantics for overlapping patterns. These two forces have lead to representing functions by case trees as in the last chapter. However, this restriction leads to some problems that we will discuss. At the end of this chapter, we will describe how these problems can be solved by allowing definitions with more general pattern sets.

Remark that the problems we try to fix are not just problems with the current approach, but consequences of the two design goals. So in order to fix the problems, we will have to give up on both goals. We argue that this is worth it from a practical perspective because it leads to definitions that are both precise and easy to understand.

5.1 Two design goals of pattern matching

When we want to use dependent pattern matching, a conflict between theory and practice presents itself. On the one hand, type theorists want definitions to be represented by case trees and to be structurally recursive, because functions defined in this way can be written in function of eliminators [GMM06]. This guarantees that these functions will not break important properties of type theory such as strong normalization or the Church-Rosser property.

On the other hand, functional programmers are used to writing definitions with overlapping patterns that follow the first-match semantics, because this reduces the number of clauses required in some cases. Hence they might think it is very restrictive and cumbersome to allow only coverings as the patterns of a function. The consequence of using first-match semantics is that clauses only hold as definitional equalities when all the previous clauses have failed to match the arguments. Hence the meaning of a clause depends on all the clauses above it, making it harder to understand.

In attempt to reconcile these two goals, Agda [Nor07] allows patterns to overlap but translates definitions internally to a case tree as described in the previous chapter. While building this case tree, first-match semantics are used to choose between overlapping

clauses. Internally, Agda will continue to use this case tree to represent the function, not the original definition.

5.2 Problem statement

The differences between the original definition and the case tree lead to three problems: the first-match semantics don't capture the true meaning of overlapping patterns, clauses don't hold as definitional equalities, and the semantics of a definition depend on the order of the clauses.

5.2.1 First-match semantics for overlapping patterns

If the patterns in a definition overlap, then the first-match semantics dictate that the first matching clause is always used. In the translation to a case tree, this is done by splitting the clauses and then throwing away some of the results. For example, consider the following (somewhat artificial) definition of $\texttt{plus} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$:

$$\begin{array}{llll} \texttt{plus} & \texttt{zero} & y & = & y \\ \texttt{plus} & (\texttt{suc}\ x) & \texttt{zero} & = & \texttt{suc}\ x \\ \texttt{plus} & x & (\texttt{suc}\ y) & = & \texttt{suc}\ (\texttt{plus}\ x\ y) \end{array} \tag{5.1}$$

To build a case tree, we start in the root node with label $x\ y$ where $x, y : \texttt{Nat}$. The first step is to split this pattern on the variable x. This will specialize the third clause into two clauses:

$$\begin{array}{llll} \texttt{plus} & \texttt{zero} & (\texttt{suc}\ y) & = & \texttt{suc}\ (\texttt{plus}\ \texttt{zero}\ y) \\ \texttt{plus} & (\texttt{suc}\ x) & (\texttt{suc}\ y) & = & \texttt{suc}\ (\texttt{plus}\ (\texttt{suc}\ x)\ y) \end{array} \tag{5.2}$$

The resulting case tree will look as follows:

$$x\ y \quad x : \texttt{Nat} \begin{cases} \texttt{zero}\ y \mapsto y \\ (\texttt{suc}\ x)\ y \quad y : \texttt{Nat} \begin{cases} (\texttt{suc}\ x)\ \texttt{zero} \mapsto \texttt{suc}\ x \\ (\texttt{suc}\ x)\ (\texttt{suc}\ y) \mapsto \texttt{suc}\ (\texttt{plus}\ (\texttt{suc}\ x)\ y) \end{cases} \end{cases}$$

The first of the new clauses is not necessary to build the case tree, so it is discarded. Is this case tree still equivalent to the original definition when we discard this clause? Suppose we want to evaluate $\texttt{plus}\ x\ (\texttt{suc}\ y)$ where x and y are free variables. Then the splitting tree contains no pattern matching these arguments, while the original definition did. So if we want to keep as many reduction rules as possible, then the answer is no: the case tree is not equivalent. Indeed, even if the patterns form a covering matching can still fail if the arguments contain free variables. So it can be useful to keep overlapping patterns.

As another example, let's look at the following (more standard) definition of `plus`:

$$\begin{array}{llll} \texttt{plus} & \texttt{zero} & y & = & y \\ \texttt{plus} & (\texttt{suc}\ x) & y & = & \texttt{suc}\ (\texttt{plus}\ x\ y) \end{array} \tag{5.3}$$

Then there are no possible evaluation steps of the term `plus` x (`suc zero`), even though we would like to reduce it to `suc` x. We can add the following two clauses

$$\begin{array}{lllll} \texttt{plus} & x & \texttt{zero} & = & x \\ \texttt{plus} & x & (\texttt{suc}\ y) & = & \texttt{suc}\ (\texttt{plus}\ x\ y) \end{array} \tag{5.4}$$

that allow us to conclude `plus` x (`suc zero`) $\to$ `suc` (`plus` x `zero`) $\to$ `suc` x. In practice, it can sometimes be very useful to treat overlapping patterns in this way instead of using the first-match semantics.

5.2.2 Definitional equalities are lost

Since not all complete function definitions have an equivalent case tree, sometimes clauses have to be split while building the case tree. This is even the case when the patterns of the definition do not overlap. This is illustrated by the following definition of the function `majority` : `Bool` $\to$ `Bool` $\to$ `Bool` $\to$ `Bool` due to Gérard Berry:

$$\begin{array}{llllll} \texttt{majority} & \texttt{true} & \texttt{true} & \texttt{true} & = & \texttt{true} \\ \texttt{majority} & x & \texttt{false} & \texttt{true} & = & x \\ \texttt{majority} & \texttt{true} & y & \texttt{false} & = & y \\ \texttt{majority} & \texttt{false} & \texttt{true} & z & = & z \\ \texttt{majority} & \texttt{false} & \texttt{false} & \texttt{false} & = & \texttt{false} \end{array} \tag{5.5}$$

The patterns of this definition are complete: for all concrete values of the booleans x, y, z there is a clause that defines `majority` $x\ y\ z$. However, they do not form a covering. When creating a case tree from this definition, the second and third clauses are specialized into two clauses each. This results in the following case tree:

$$\begin{array}{l} x\ y\ z \\ \quad x : \texttt{Bool} \left\{ \begin{array}{l} \texttt{true}\ y\ z \\ \quad\quad y : \texttt{Bool} \left\{ \begin{array}{l} \texttt{true true}\ z \\ \quad\quad\quad z : \texttt{Bool} \left\{ \begin{array}{l} \texttt{true true true} \mapsto \texttt{true} \\ \texttt{true true false} \mapsto \texttt{true} \end{array} \right. \\ \texttt{true false}\ z \\ \quad\quad\quad z : \texttt{Bool} \left\{ \begin{array}{l} \texttt{true false true} \mapsto \texttt{true} \\ \texttt{true false false} \mapsto \texttt{false} \end{array} \right. \end{array} \right. \\ \texttt{false}\ y\ z \\ \quad\quad y : \texttt{Bool} \left\{ \begin{array}{l} \texttt{false true}\ z \mapsto z \\ \texttt{false false}\ z \\ \quad\quad\quad z : \texttt{Bool} \left\{ \begin{array}{l} \texttt{false false true} \mapsto \texttt{false} \\ \texttt{false false false} \mapsto \texttt{false} \end{array} \right. \end{array} \right. \end{array} \right. \end{array}$$

On first sight, this case tree seems to be equivalent with the original definition. However, its evaluation rules are slightly different. If x : `Bool` is a free variable, then according to the original definition `majority` x `false true` evaluates to x. According to the case tree, this is no longer true because the clause has been split into `majority true false true` = `true` and `majority false false true` = `false`. So we have lost the definitional equality `majority` x `false true` = x.

Remark that the equality still holds in both cases $x = \texttt{true}$ and $x = \texttt{false}$, hence it is possible to construct a term $(x : \texttt{Nat}) \vdash p : \texttt{majority}\ x\ \texttt{false}\ \texttt{true} \equiv_{\texttt{Nat}} x$ by pattern matching on x. But this tells us only that the terms are propositionally equal, which is a weaker kind of equality than the definitional equality.

5.2.3 Order of the patterns matters

Even when the patterns of a function do not overlap and actually form a covering, the order of the clauses can still influence the meaning of that function when it is translated to a case tree. For example, consider the following function $\texttt{p} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}$ which is a simplified version of an example given by Achim Jung on the Agda mailing list. It checks whether two given natural numbers are equal modulo 2:

$$\begin{array}{llcl} \texttt{p}\ \ \texttt{zero} & \texttt{zero} & = & \texttt{true} \\ \texttt{p}\ \ \texttt{zero} & (\texttt{suc}\ \texttt{zero}) & = & \texttt{false} \\ \texttt{p}\ \ \texttt{zero} & (\texttt{suc}\ (\texttt{suc}\ y)) & = & \texttt{p}\ \texttt{zero}\ y \\ \texttt{p}\ \ (\texttt{suc}\ \texttt{zero}) & \texttt{zero} & = & \texttt{false} \\ \texttt{p}\ \ (\texttt{suc}\ (\texttt{suc}\ x)) & \texttt{zero} & = & \texttt{p}\ x\ \texttt{zero} \\ \texttt{p}\ \ (\texttt{suc}\ x) & (\texttt{suc}\ y) & = & \texttt{p}\ x\ y \end{array} \tag{5.6}$$

Remark that there are again no overlapping patterns in this definition. In fact, the patterns even form a covering in this case. Here is a case tree that corresponds exactly to the function p:

$$\begin{array}{l} n\ m \\ \quad n : \texttt{Nat} \left\{ \begin{array}{l} \texttt{zero}\ m \\ \quad\quad m : \texttt{Nat} \left\{ \begin{array}{l} \texttt{zero}\ \texttt{zero} \mapsto \texttt{true} \\ \texttt{zero}\ (\texttt{suc}\ m) \\ \quad\quad\quad m : \texttt{Nat} \left\{ \begin{array}{l} \texttt{zero}\ (\texttt{suc}\ \texttt{zero}) \mapsto \texttt{false} \\ \texttt{zero}\ (\texttt{suc}\ (\texttt{suc}\ m)) \mapsto \texttt{p}\ \texttt{zero}\ m \end{array} \right. \end{array} \right. \\ (\texttt{suc}\ n)\ m \\ \quad\quad m : \texttt{Nat} \left\{ \begin{array}{l} (\texttt{suc}\ n)\ \texttt{zero} \\ \quad\quad\quad n : \texttt{Nat} \left\{ \begin{array}{l} (\texttt{suc}\ \texttt{zero})\ \texttt{zero} \mapsto \texttt{false} \\ (\texttt{suc}\ (\texttt{suc}\ n))\ \texttt{zero} \mapsto \texttt{p}\ n\ \texttt{zero} \end{array} \right. \\ (\texttt{suc}\ n)\ (\texttt{suc}\ m) \mapsto \texttt{p}\ n\ m \end{array} \right. \end{array} \right. \end{array}$$

However, when building a case tree from the above definition in the way described in the previous chapter, the result will be a *different* case tree. Indeed, the last pattern is split in the following two clauses:

$$\begin{array}{llcl} \texttt{p}\ \ (\texttt{suc}\ \texttt{zero}) & (\texttt{suc}\ y) & = & \texttt{p}\ \texttt{zero}\ y \\ \texttt{p}\ \ (\texttt{suc}\ (\texttt{suc}\ x)) & (\texttt{suc}\ y) & = & \texttt{p}\ (\texttt{suc}\ x)\ y \end{array}$$

If the last clause was placed first instead, then the correct covering would have been reconstructed. One possible solution for this specific problem would be to build all possible case trees and choose the one that gives the least number of extra splittings, but this could become very time-consuming for long definitions.

5.3 Allowing overlapping patterns

In order to be able to reduce definitions by pattern matching to eliminators, they are converted to case trees. During this conversion, first-match semantics are used. We have encountered the following problems caused by this conversion:

- First-match semantics cause overlapping patterns to be discarded, even in situations where it could be useful to keep them.
- Clauses must be split to create a covering, changing the reduction rules for arguments with free variables. What clauses are split depends on the order of the clauses.
- Clauses can even be split when the patterns already form a covering, depending on the order of the clauses. This results in a different covering than the one we started from.

All these problems come down to the fact that not all clauses are preserved as definitional equalities in the translation to a case tree. In Agda, this means that the clauses given by the programmer do not correspond to the clauses used internally, which is annoying and can lead to unexpected results.

To fix these problems, we will extend pattern matching in order to allow more general pattern sets than just coverings. In particular, the patterns in a definition will be allowed 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 it can't be decided whether a pattern matches the arguments or not. Instead, all patterns are considered for pattern matching in parallel. If one of the patterns matches the arguments, evaluation continues; even if another pattern blocks. 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 the three above problems. Overlapping patterns are allowed, hence there is no need to discard them. We don't need patterns to form a covering, so there is no need to split clauses. We don't have to translate definitions to case trees, so the meaning of a definition does not depend on order of the patterns. The result is that definitions by pattern matching feel more like mathematical definitions, rather than program instructions.

Our approach also has some drawbacks:

- First of all, we lose the ability to translate definitions to pure type theory with eliminators. To guarantee correctness (completeness, termination, and confluence), we will thus need to reason about the definitions directly. The criterion for termination we gave in the previous chapter doesn't depend on the fact that the patterns form a covering, but the criteria for completeness and confluence do. In the next chapter, we will describe how to check completeness and confluence in the presence of overlapping patterns.
- We also lose the first-match semantics. This doesn't restrict the definitions we can give, but it requires us to write longer definitions in some cases. This is an unavoidable problem because we want clauses to be order-independent.

- Finally, we lose the ability to represent functions by case trees, hence the ability to evaluate functions efficiently. In the next chapter, we will extend case trees with *catchall subtrees* which allows us to represent these more general definitions by case trees.

Chapter 6

Checking definitions with overlapping patterns

In the last chapter, we proposed a generalized form of pattern matching with dependent types. However, the standard techniques are not sufficient to check correctness of these generalized definitions. In particular, the criterion for completeness has to be adapted and we also need a criterion for confluence. We will describe both in this chapter. We will also present how these generalized definitions can be represented by case trees by adding catchall subtrees.

6.1 Checking completeness

We want to allow more general pattern sets than just coverings, but at the same time, we still want to be sure that the pattern set is complete in the sense of definition 3.11. So far, coverings are the only pattern sets we know that are guaranteed to be complete. The idea is that we start from a covering and then change it step by step while preserving completeness. More concretely, we can make the following changes to a set of patterns without losing completeness:

- We can add new patterns.
- We can replace a pattern p by a more general pattern p' such that $p' \supseteq p$, i.e. all terms that match p also match p'.
- We can *contract* a number of patterns $p_1, \ldots, p_n$ to a pattern p where $p \supseteq p_i$ for $i = 1, \ldots, n$.

Starting from a covering and repeating these three steps, we can build a large number of overlapping pattern sets; in particular all pattern sets of the examples from the previous chapter can be constructed in this way.

These three rules can be nicely summarized in one criterion:

Theorem 6.1. *Let P be a set of lists of patterns of the same type. If there exists a covering O such that for each $\bar{q} \in O$, there exists a $\bar{p} \in P$ such that $\bar{p} \supseteq \bar{q}$, then P is complete.*

Proof. Since the covering O is complete by theorem 4.8, each closed term $\bar{t}$ matches a $\bar{q} \in O$, i.e. there exists a substitution σ such that $\bar{t} = \lceil \bar{q} \rceil \sigma$. By assumption, there exists a $\bar{p}$ such that $\bar{p} \supseteq \bar{q}$, i.e. there exists a substitution δ such that $\lceil \bar{p} \rceil \delta = \lceil \bar{q} \rceil$. Then we have $\bar{t} = \lceil \bar{p} \rceil \delta\sigma$, hence the set of patterns P is complete. □

This leaves us with the question of how to find this covering O. But in fact, we already know how: by building a case tree from the patterns in P. If we succeed in building a case tree, then the patterns at the leaves of the case tree form a covering O. We claim that this covering satisfies the conditions of the theorem. Indeed, while building the case tree we only split patterns in P or we discard overlapping patterns. Hence for each pattern $\bar{q}$ in the resulting covering O, there exists a pattern $\bar{p}$ in P such that $\bar{p} \supseteq \bar{q}$, exactly as we wanted.

6.2 Checking confluence

To check the confluence of a definition, we have to check for each pair of clauses that whenever a term matches both their patterns, then they give the same result for that term. If there are no terms matching both patterns, then this is trivially satisfied. So the first thing we need to check is whether two patterns overlap, i.e. whether there is a term that matches both patterns. We make the following observation: let $\bar{p}_1$ and $\bar{p}_2$ be two patterns that have a most general unifier σ, so that $\lceil \bar{p}_1 \rceil \sigma = \lceil \bar{p} \rceil = \lceil \bar{p}_2 \rceil \sigma$. 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.

Now suppose we have found two clauses $f\ \bar{p}_1 = t_1$ and $f\ \bar{p}_2 = t_2$ that overlap, i.e. $\lceil \bar{p}_1 \rceil \sigma = \lceil \bar{p}_2 \rceil \sigma$ for some most general unifier σ. In order to obtain confluence, we require that the right-hand sides t_1 and t_2 are equal under this substitution σ. But what kind of equality do we want? Let's look at the following two overlapping clauses of the function `plus`:

$$\begin{array}{llll} \texttt{plus} & (\texttt{suc}\ x_1) & y_1 & = \ \texttt{suc}\ (\texttt{plus}\ x_1\ y_1) \\ \texttt{plus} & x_2 & (\texttt{suc}\ y_2) & = \ \texttt{suc}\ (\texttt{plus}\ x_2\ y_2) \end{array}$$

We have numbered the variables in order to avoid confusion. The most general unifier of the patterns is $\sigma = [x_2 \mapsto \texttt{suc}\ x_1, y_1 \mapsto \texttt{suc}\ y_2]$. If we apply this substitution to the right-hand sides, we get $\texttt{suc}\ (\texttt{plus}\ x_1\ (\texttt{suc}\ y_2))$ for the first clause and $\texttt{suc}\ (\texttt{plus}\ (\texttt{suc}\ x_1)\ y_2)$ for the second clause. These are certainly not syntactically equal, but they do evaluate to the same normal form $\texttt{suc}\ (\texttt{suc}\ (\texttt{plus}\ x_1\ y_2))$. Hence they are *definitionally equal*. This is the criterion we will use for checking confluence. In section 7.2, we will see that it is *not* sufficient to ask that $t_1\sigma$ and $t_2\sigma$ are only propositionally equal.

Theorem 6.2. *Let C be a set of clauses for the function $f : \Phi \to T$. If 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 $\bar{p}_2$ with all pattern variables as flexible variables*

- *either succeeds positively with result σ and $t_1\sigma$ is definitionally equal to $t_2\sigma$,*
- *or succeeds negatively,*

then the clauses C are confluent.

Proof. Let $\bar{u} : \Phi$ and suppose $f\ \bar{u} \longrightarrow_C v_1$ and $f\ \bar{u} \longrightarrow_C v_2$. By definition of the evaluation rule $\longrightarrow_C$, there exist clauses $f\ \bar{p}_1 = t_1$, $f\ \bar{p}_2 = t_2$ in C and substitutions δ_1, δ_2 such that $\lceil \bar{p}_1 \rceil \delta_1 = \bar{u} = \lceil \bar{p}_2 \rceil \delta_2$, $t_1\delta_1 = v_1$ and $t_2\delta_2 = v_2$. In particular we have that $\delta = \delta_1; \delta_2$ is a unifier of $\bar{p}_1$ and $\bar{p}_2$, so unification of $\bar{p}_1$ and $\bar{p}_2$ cannot succeed negatively. By the assumption of the theorem, unification of $\bar{p}_1$ and $\bar{p}_2$ must succeed positively with result σ and there must exist a term t such that $t_1\sigma \longrightarrow_C^* t$ and $t_2\sigma \longrightarrow_C^* t$. Because σ is a most general unifier, there exists a substitution δ' such that $\delta = \sigma; \delta'$. This implies $v_1 = t_1\delta = (t_1\sigma)\delta' \longrightarrow_C^* t\delta'$ and $v_2 = t_2\delta = (t_2\sigma)\delta' \longrightarrow_C^* t\delta'$, hence the set of clauses C is confluent. □

Now how can this criterion be used to implement a confluence checker? Suppose we have a function $f : \Phi \to T$ and two clauses $f\ \bar{p}_1 = t_1$ and $f\ \bar{p}_2 = t_2$ where $\Gamma|\Delta_1 \vdash \bar{p}_1 : \Phi$ **pattern** and $\Gamma|\Delta_2 \vdash \bar{p}_2 : \Phi$ **pattern**. To check confluence of these clauses, the algorithm proceeds as follows:

First, we attempt to unify $\bar{p}_1$ and $\bar{p}_2$. If the unification succeeds negatively, then the clauses are confluent. If the unification succeeds positively with result σ, then we apply σ to t_1 and t_2. We proceed by evaluating $t_1\sigma$ and $t_2\sigma$ to normal form. If the results are syntactically equal, then the clauses are confluent. The algorithm is summarized in figure 6.1.

$$\frac{\textsc{unify}(\lceil \bar{p}_1 \rceil, \lceil \bar{p}_2 \rceil) \Rightarrow \sigma \qquad \Gamma\Delta_1\Delta_2 \vdash t_1\sigma = t_2\sigma : T[\Phi \mapsto \lceil \bar{p}_1 \rceil]\sigma}{\textsc{Confluent}(f\ \bar{p}_1 = t_1, f\ \bar{p}_2 = t_2)}$$

$$\frac{\textsc{unify}(\lceil \bar{p}_1 \rceil, \lceil \bar{p}_2 \rceil) \Uparrow}{\textsc{Confluent}(f\ \bar{p}_1 = t_1, f\ \bar{p}_2 = t_2)}$$

Figure 6.1: The algorithm for checking confluence. Since σ is a unifier of $\lceil \bar{p}_1 \rceil$ and $\lceil \bar{p}_2 \rceil$, we have $T[\Phi \mapsto \lceil \bar{p}_1 \rceil]\sigma = T[\Phi \mapsto \lceil \bar{p}_2 \rceil]\sigma$; i.e. the terms we compare have equal type.

Remark that in order to check confluence, we already need to evaluate the function for which we are checking confluence. Hence we can only check confluence after we have first checked termination of the definition.

There are two ways in which the confluence check can fail:

- The unification of the patterns can fail. Remark that unification of patterns consisting of only constructors and variables always succeeds (either positively or negatively). So the unification can only fail if we have to unify an inaccessible pattern with a constructor pattern or another inaccessible pattern.
- The normal forms of $t_1\sigma$ and $t_2\sigma$ may not be syntactically equal. Either this is because the clauses are not confluent, or they are confluent but a simple syntactic check of the normal forms is insufficient to see this.

In our experience, the first failure occurs more often than the second one. So in order to improve the algorithm, it would probably be more fruitful to improve the unification algorithm rather than the comparison of the right-hand sides.

6.3 Case trees for overlapping clauses

By allowing patterns to overlap, we have lost the ability to represent function definitions as case trees. In this section, we will discuss how functions whose patterns do not form a covering can be represented in a similar way. This approach has been inspired by the paper of A. Gräf of left-to-right tree pattern matching [Gr91].

To capture the extra reduction rules given by overlapping patterns, we will add a catchall subtree to each internal node of a case tree. This fallback subtree captures exactly the information from the user patterns that is lost in the translation to a case tree. During evaluation, a catchall subtree will give an alternative when the matching with the main case tree is inconclusive.

A catchall subtree can be recognized by the fact that it is always labeled with the same patterns as its parent node. For example, here is a case tree with a catchall subtree for the overlapping definition of `plus`:

$$
\begin{array}{l}
n\ m \\
\quad n : \texttt{Nat} \left\{ \begin{array}{l}
\texttt{zero}\ m \mapsto m \\
(\texttt{suc}\ n)\ m \mapsto \texttt{suc}\ (\texttt{plus}\ n\ m) \\
n\ m \\
\quad m : \texttt{Nat} \left\{ \begin{array}{l} n\ \texttt{zero} \mapsto n \\ n\ (\texttt{suc}\ m) \mapsto \texttt{suc}\ (\texttt{plus}\ n\ m) \end{array} \right.
\end{array} \right.
\end{array}
$$

Catchall subtrees are not required to be complete, i.e. the patterns of the children of a node are no longer required to form a covering. For example, here is a case tree with catchall subtrees that captures the exact semantics of the `majority` function 5.5.

$$
\begin{array}{l}
x\ y\ z \\
\quad x : \texttt{Bool} \left\{ \begin{array}{l}
\texttt{true}\ y\ z \\
\quad y : \texttt{Bool} \left\{ \begin{array}{l}
\texttt{true}\ \texttt{true}\ z \\
\quad z : \texttt{Bool} \left\{ \begin{array}{l} \texttt{true}\ \texttt{true}\ \texttt{true} \mapsto \texttt{true} \\ \texttt{true}\ \texttt{true}\ \texttt{false} \mapsto \texttt{true} \end{array} \right. \\
\texttt{true}\ \texttt{false}\ z \\
\quad z : \texttt{Bool} \left\{ \begin{array}{l} \texttt{true}\ \texttt{false}\ \texttt{true} \mapsto \texttt{true} \\ \texttt{true}\ \texttt{false}\ \texttt{false} \mapsto \texttt{false} \end{array} \right. \\
\texttt{true}\ y\ z \\
\quad z : \texttt{Bool} \left\{ \begin{array}{l} \texttt{true}\ y\ \texttt{false} \mapsto y \end{array} \right.
\end{array} \right. \\
\texttt{false}\ y\ z \\
\quad y : \texttt{Bool} \left\{ \begin{array}{l}
\texttt{false}\ \texttt{true}\ z \mapsto z \\
\texttt{false}\ \texttt{false}\ z \\
\quad z : \texttt{Bool} \left\{ \begin{array}{l} \texttt{false}\ \texttt{false}\ \texttt{true} \mapsto \texttt{false} \\ \texttt{false}\ \texttt{false}\ \texttt{false} \mapsto \texttt{false} \end{array} \right.
\end{array} \right. \\
x\ y\ z \\
\quad y : \texttt{Bool} \left\{ \begin{array}{l}
x\ \texttt{false}\ z \\
\quad z : \texttt{Bool} \left\{ \begin{array}{l} x\ \texttt{false}\ \texttt{true} \mapsto x \end{array} \right.
\end{array} \right.
\end{array} \right.
\end{array}
$$

Remark also that while extra clauses have been added in this case tree, all original clauses are still present. The added clauses are merely specialized versions of the original clauses that are needed to get a complete case tree when we remove the catchall subtrees.

Definition 6.3 (Catchall tree). *A* catchall tree *for a function $f : \Delta \to T$ with label $\bar{p} : \Delta$* ***pattern*** *is one of the following*

- *Either it is an internal node and there is a pattern variable x of $\bar{p}$ such that the subtrees are again catchall trees with labels $\bar{p}_i$ where $\bar{p} \Rightarrow^{x}_{c_i} \bar{p}_i$ for constructors c_i.*
- *or it is a leaf node containing a term $t : T[\bar{x} \mapsto \lceil \bar{p} \rceil]$ called the right-hand side.*
- *or $\bar{p}$ contains an absurd pattern $\emptyset$, in this case the tree is a leaf node with no right-hand side.*

A *case tree with catchall subtrees* is the same as a regular case tree, except that each internal node with label $\bar{p}$ is allowed to have an additional subtree that is a catchall tree with the same label $\bar{p}$. Each internal node of a catchall subtree is also allowed to have an additional catchall subtree, and so on.

Case trees with catchall subtrees can be built from a (complete) set of clauses in almost the same way as regular case trees. When building the subtrees, we will take care not to throw away any information about the reduction rules contained in the clauses. Remember that while building a case tree, the set of clauses was partitioned in sets C_j for each constructor c_j, and two additional sets C_{var} and C_{rest}. The clauses that are lost as definitional equalities in the translation to a case tree are exactly the ones in C_{var} and C_{rest}. Hence these clauses will be used to build the catchall subtree. Remark that the variable x we chose to build the covering is no longer a blocking variable for the clauses $C_{var} \cup C_{rest}$, so a different variable will have to be chosen in the catchall subtree.

Let us give an example of how a case tree with catchall subtrees is built. Define a data type $x \stackrel{?}{=}_A y$ with one parameter $A : Set$, two indices $x, y : A$ and two constructors:

$$\begin{array}{rcl} \mathtt{eq} & : & (x : A) \to x \stackrel{?}{=}_A x \\ \mathtt{neq} & : & (x : A)(y : A) \to x \stackrel{?}{=}_A y \end{array} \tag{6.1}$$

Here is the definition of a function $f : (y : \mathtt{Bool})(p : \mathtt{true} \stackrel{?}{=}_{\mathtt{Bool}} y) \to \mathtt{Bool}$:

$$\begin{array}{llll} f & \mathtt{true} & p & = \mathtt{true} \\ f & y & (\mathtt{neq}\ \lfloor \mathtt{true} \rfloor\ \lfloor y \rfloor) & = y \end{array} \tag{6.2}$$

The first step in building a case tree is to split on the variable $y : \mathtt{Bool}$. This partitions the clauses into the following sets:

$$\begin{array}{lcl} C_{\mathtt{true}} & = & \{f\ \mathtt{true}\ p = \mathtt{true}\} \\ C_{\mathtt{false}} & = & \emptyset \\ C_{var} & = & \{f\ y\ (\mathtt{neq}\ \lfloor \mathtt{true} \rfloor\ \lfloor y \rfloor) = y\} \\ C_{rest} & = & \emptyset \end{array}$$

The case tree is built as normal, the only interesting part is that the clause in C_{var} is used to build a catchall subtree. Here is the final result:

$$\begin{array}{l} y\ p \\ \quad y : \mathtt{Bool} \left\{ \begin{array}{l} \mathtt{true}\ p \mapsto \mathtt{true} \\ \mathtt{false}\ p \\ \quad p : \mathtt{true} \stackrel{?}{=}_{\mathtt{Bool}} \mathtt{false} \left\{ \begin{array}{l} \mathtt{false}\ (\mathtt{neq}\ \lfloor \mathtt{true} \rfloor\ \lfloor \mathtt{false} \rfloor) \mapsto \mathtt{false} \end{array} \right. \\ y\ p \\ \quad p : \mathtt{true} \stackrel{?}{=}_{\mathtt{Bool}} y \left\{ \begin{array}{l} y\ (\mathtt{neq}\ \lfloor \mathtt{true} \rfloor\ \lfloor y \rfloor) \mapsto y \end{array} \right. \end{array} \right. \end{array}$$

Chapter 7

Evaluation

In this chapter, we will give some examples of definitions with overlapping patterns. By defining functions with overlapping patterns, we get extra evaluation rules, which sometimes makes it easier to prove propositions that mention these functions. We will also give an example where our confluence check fails unexpectedly. We will discuss an alternative criterion for confluence that does accept this definition. However, we will argue that this alternative criterion is ultimately not enough to obtain the kind of confluence we need, hence we will stick to our original criterion.

7.1 Some examples

7.1.1 Addition of natural numbers

We have mentioned it often enough, but here is the full definition of $\texttt{plus} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$ with overlapping patterns one more time:

$$\begin{array}{llll} \texttt{plus} & \texttt{zero} & y & = & y \\ \texttt{plus} & (\texttt{suc}\ x) & y & = & \texttt{suc}\ (\texttt{plus}\ x\ y) \\ \texttt{plus} & x & \texttt{zero} & = & x \\ \texttt{plus} & x & (\texttt{suc}\ y) & = & \texttt{suc}\ (\texttt{plus}\ x\ y) \end{array} \tag{7.1}$$

Remark that the reduction rules of this definition are maximal in some sense: whenever there appears a constructor in any of the arguments, we can always immediately apply one of the clauses. We could add more clauses to the definition, for example $\texttt{plus}\ (\texttt{suc}\ m)\ (\texttt{suc}\ n) = \texttt{suc}\ (\texttt{suc}\ (\texttt{plus}\ m\ n))$, but this doesn't add any more power to the definition. Indeed, we can get the same evaluation rule by combining the second and the fourth clause.

In section 6.2, we have shown how to check confluence of the second and the fourth clause. The other pairs can be checked in a similar way.

Using this definition of `plus` can make it a lot easier to define new functions where natural numbers appear in the types. For example, we can readily define the function $\texttt{plus-comm} : (m : \texttt{Nat})(n : \texttt{Nat}) \to (\texttt{plus}\ m\ n) \equiv_{\texttt{Nat}} (\texttt{plus}\ n\ m)$ that proves the commutativity of `plus` as follows:

$$\begin{array}{lll} \texttt{plus-comm} & \texttt{zero} & n = \texttt{refl}\ n \\ \texttt{plus-comm} & (\texttt{suc}\ m) & n = \texttt{cong}\ \texttt{suc}\ (\texttt{plus-comm}\ m\ n) \end{array} \tag{7.2}$$

Here, `cong` is a congruence rule that has type $(f : A \to B)(x : A)(y : A) \to x \equiv_A y \to f\ x \equiv_A f\ y$. It can be defined by pattern matching as

$$\texttt{cong}\ f\ \lfloor x \rfloor\ \lfloor x \rfloor\ (\texttt{refl}\ x) = \texttt{refl}\ (f\ x)$$

To give this proof for the non-overlapping definition of `plus`, the Agda standard library first needs a lemma to prove that $1 + (m + n) \equiv m + (1 + n)$, and then the proof itself still takes approximately eight lines.

7.1.2 Operations on booleans

Analogously to the definition of `plus`, we can define operations `and` and `or` with overlapping patterns:

$$\begin{array}{llll} \texttt{and} & \texttt{true} & y & = y \\ \texttt{and} & \texttt{false} & y & = \texttt{false} \\ \texttt{and} & x & \texttt{true} & = x \\ \texttt{and} & x & \texttt{false} & = \texttt{false} \end{array} \tag{7.3}$$

$$\begin{array}{llll} \texttt{or} & \texttt{true} & y & = \texttt{true} \\ \texttt{or} & \texttt{false} & y & = y \\ \texttt{or} & x & \texttt{true} & = \texttt{true} \\ \texttt{or} & x & \texttt{false} & = x \end{array} \tag{7.4}$$

Confluence can be checked very easily for these two definitions.

7.1.3 Concatenation of vectors

We can also give a definition of the concatenation `concat` of two vectors that has type $\texttt{concat} : (m : \texttt{Nat})(n : \texttt{Nat}) \to \texttt{Vec}\ A\ m \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{plus}\ m\ n)$:

$$\begin{array}{llllll} \texttt{concat} & \lfloor \texttt{zero} \rfloor & n & \epsilon & w & = w \\ \texttt{concat} & m & \lfloor \texttt{zero} \rfloor & v & \epsilon & = v \\ \texttt{concat} & \lfloor \texttt{suc}\ m \rfloor & n & (\texttt{cons}\ m\ a\ v) & w & = \texttt{cons}\ m\ a\ (\texttt{concat}\ m\ n\ v\ w) \end{array} \tag{7.5}$$

The non-standard clause in this definition is the second one, it overlaps with both the first clause and the third clause. In both cases, it can be checked that the clauses satisfy the criterion for confluence. It is also interesting to 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 in an essential way upon the fact that the definition of `plus` has overlapping clauses.

7.1.4 Transitivity of the propositional equality

Remember that 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 by pattern matching. For example, here is a definition of $\texttt{trans} : (x : A)(y : A)(z : A) \to x \equiv_A y \to y \equiv_A z \to x \equiv_A z$:

$$\begin{array}{llllll} \texttt{trans} & \lfloor y \rfloor & \lfloor y \rfloor & z & (\texttt{refl}\ y) & q & = q \\ \texttt{trans} & x & \lfloor y \rfloor & \lfloor y \rfloor & p & (\texttt{refl}\ y) & = p \end{array} \tag{7.6}$$

Remark that the patterns overlap; their unification is given by $[x \mapsto y, z \mapsto y, p \mapsto \texttt{refl}\ y, q \mapsto \texttt{refl}\ y]$. Under this substitution, the right-hand sides are equal, hence the clauses are confluent. This shows that the confluence checker also works correctly in the presence of inaccessible patterns.

Under the interpretation of proofs of $x \equiv_A y$ as paths from x to y in homotopy type theory, these two clauses tell us that if we compose a path p with the identity path on either side, the result will be homotopic to p.

7.1.5 A counterexample: multiplication of natural numbers

Let us look at another function on natural numbers, multiplication:

$$\begin{array}{llll} \texttt{mult}\ \texttt{zero} & y & = & \texttt{zero} \\ \texttt{mult}\ (\texttt{suc}\ x) & y & = & \texttt{plus}\ (\texttt{mult}\ x\ y)\ y \\ \texttt{mult}\ x & \texttt{zero} & = & \texttt{zero} \\ \texttt{mult}\ x & (\texttt{suc}\ y) & = & \texttt{plus}\ x\ (\texttt{mult}\ x\ y) \end{array} \tag{7.7}$$

Let us focus on the confluence of the second and the fourth clause. If we rename the variables as $\texttt{mult}\ (\texttt{suc}\ x_1)\ y_1 = \texttt{plus}\ (\texttt{mult}\ x_1\ y_1)\ y_1$ and $\texttt{mult}\ x_2\ (\texttt{suc}\ y_2) = \texttt{plus}\ x_1\ (\texttt{mult}\ x_1\ y_1)$, then unification of the patterns gives $\sigma = [x_2 \mapsto \texttt{suc}\ x_1, y_1 \mapsto \texttt{suc}\ y_2]$. Under this substitution, the right-hand sides become

$$\texttt{plus}\ (\texttt{mult}\ x_1\ (\texttt{suc}\ y_2))\ (\texttt{suc}\ y_2) \longrightarrow \texttt{suc}\ (\texttt{plus}\ (\texttt{plus}\ x_1\ (\texttt{mult}\ x_1\ y_2))\ y_2)$$

and

$$\texttt{plus}\ (\texttt{suc}\ x_1)\ (\texttt{mult}\ (\texttt{suc}\ x_1)\ y_2) \longrightarrow \texttt{suc}\ (\texttt{plus}\ x_1\ (\texttt{plus}\ (\texttt{mult}\ x_1\ y_2)\ y_2))$$

We see that the right-hand sides do not have the same normal form when we apply the substitution, so they are not definitionally equal. Hence this definition of `mult` does not satisfy the criterion for confluence.

By first proving that `plus` is associative, it is possible to give a term

$$\begin{array}{l} (x : \texttt{Nat})(y : \texttt{Nat}) \vdash \\ \quad p : \texttt{suc}\ (\texttt{plus}\ (\texttt{plus}\ x\ (\texttt{mult}\ x\ y))\ y) \equiv_{\texttt{Nat}} \texttt{suc}\ (\texttt{plus}\ x\ (\texttt{plus}\ (\texttt{mult}\ x\ y)\ y)) \end{array}$$

that proves that the right-hand sides are *propositionally* equal. In the next section, we will consider whether it is sufficient to have such a propositional equality instead of a definitional equality.

7.2 Explicit proofs of confluence

Instead of trying to check confluence automatically, we could ask the user for an explicit proof of confluence. We could represent a proof of confluence of two clauses $f\ \bar{p}_1 = t_1$ and $f\ \bar{p}_2 = t_2$ as a term of type $\Gamma \to \lceil \bar{p}_1 \rceil \equiv \lceil \bar{p}_2 \rceil \to t_1 \equiv t_2$, where Γ is a context containing the pattern variables of $\bar{p}_1$ and $\bar{p}_2$. For example, to prove the confluence of the clauses $\texttt{plus}\ \texttt{zero}\ y = y$ and $\texttt{plus}\ x\ \texttt{zero} = x$, we would have to give a term p of type $(x : \texttt{Nat})(y : \texttt{Nat}) \to \texttt{zero} \equiv_{\texttt{Nat}} x \to y \equiv_{\texttt{Nat}} \texttt{zero} \to x \equiv_{\texttt{Nat}} y$, which is not too hard:

$$p\ \lfloor\texttt{zero}\rfloor\ \lfloor\texttt{zero}\rfloor\ (\texttt{refl}\ \lfloor\texttt{zero}\rfloor)\ (\texttt{refl}\ \lfloor\texttt{zero}\rfloor) = \texttt{refl}\ \texttt{zero}$$

In general, when the criterion for confluence is satisfied, it is never hard to give a similar proof of confluence. But is it *enough* to have such a proof?

Let's take another look at the multiplication of natural numbers. We remarked before that it is possible to prove that this definition is confluent. But what happens if we try to evaluate `mult` (`suc` x) (`suc` y) where x and y are free variables? If we apply the second clause first, we get

$$\begin{aligned}&\texttt{mult}\ (\texttt{suc}\ x)\ (\texttt{suc}\ y)\\ &\quad\longrightarrow \texttt{plus}\ (\texttt{mult}\ x\ (\texttt{suc}\ y))\ (\texttt{suc}\ y)\\ &\quad\longrightarrow \texttt{plus}\ (\texttt{plus}\ x\ (\texttt{mult}\ x\ y))\ (\texttt{suc}\ y)\\ &\quad\longrightarrow \texttt{suc}\ (\texttt{plus}\ (\texttt{plus}\ x\ (\texttt{mult}\ x\ y))\ y)\end{aligned}$$

If we apply the fourth clause first instead, we get

$$\begin{aligned}&\texttt{mult}\ (\texttt{suc}\ x)\ (\texttt{suc}\ y)\\ &\quad\longrightarrow \texttt{plus}\ (\texttt{suc}\ x)\ (\texttt{mult}\ (\texttt{suc}\ x)\ y)\\ &\quad\longrightarrow \texttt{plus}\ (\texttt{suc}\ x)\ (\texttt{plus}\ (\texttt{mult}\ x\ y)\ y)\\ &\quad\longrightarrow \texttt{suc}\ (\texttt{plus}\ x\ (\texttt{plus}\ (\texttt{mult}\ x\ y)\ y))\end{aligned}$$

which is equal *up to associativity* of `plus`. But associativity is not a definitional equality, hence these are two different normal forms of the same term, violating the Church-Rosser property. It is true that both clauses will give the same result for *closed* arguments, but we also need confluence for arguments with free variables. Indeed, the definition of the Church-Rosser property is not limited to just closed terms. Hence the criterion we gave that uses definitional equality is the correct one, because it also guarantees confluence for *open* arguments.

Chapter 8

Link with non-overlapping definitions

We have shown that overlapping function definitions can be very useful, but we also have to worry whether they are not *too* powerful: we want our addition to be consistent with the core type theory of UTT. Specifically, we do not want to introduce new closed terms of types that are supposed to be empty (such as $\bot$). 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, there is no hope to proceed in this way.

It is not in scope of this thesis to give a full consistency proof, as consistency is usually the hardest property to prove about any version of type theory. However, we will 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. Hence we will define an equivalence $\approx$ which is *extensional*:

Definition 8.1 (Weak equivalence). *We define an equivalence relation $\approx$ on terms called* weak equivalence *as the smallest equivalence relation satisfying the following rules:*

- *If $\Gamma \vdash t_1 = t_2 : T$, then $t_1 \approx t_2$.*
- *If $f_1\ a \approx f_2\ a$ for all closed terms a, then $f_1 \approx f_2$.*
- *If $f_1 \approx f_2$ and $t_1 \approx t_2$, then $f_1\ t_1 \approx f_2\ t_2$.*

Remark that the functions f_1 and f_2 in the third rule can be dependent functions, hence the types of $f_1\ t_1$ and $f_2\ t_2$ can be different even if f_1 and f_2 have the same type. Hence $\approx$ is a *heterogeneous* equivalence relation: it is possible that $t_1 \approx t_2$ when t_1 and t_2 do not have the same type.

We can now formulate the theorem that gives the link between definitions with and without overlapping clauses:

Theorem 8.2. *If a function $f : (\bar{x} : \Delta) \to T$ is defined by a set of overlapping clauses that satisfy the criteria for completeness (6.1), termination (4.11), and confluence (6.2); then there exists a function $f' : (\bar{x} : \Delta) \to T$ whose patterns form a covering such that $f \approx f'$.*

In order to prove the theorem, we first need lemma 8.3.

Lemma 8.3. *Let t_1, t_2 be two terms with free variables in the context Γ. We have that $t_1 \approx t_2$ if and only if $t_1[\Gamma \mapsto \bar{a}] \approx t_2[\Gamma \mapsto \bar{a}]$ for all closed terms $\bar{a} : \Gamma$.*

Proof of the lemma. First, we prove that $t_1 \approx t_2$ whenever $t_1[\Gamma \mapsto \bar{a}] \approx t_2[\Gamma \mapsto \bar{a}]$ for all closed terms $\bar{a} : \Gamma$ by induction on the length of the context Γ. If $\Gamma = \epsilon$, then the lemma is trivial. So suppose $\Gamma = (x : T)\Gamma'$ and let $a : T$ be a closed term. Then $t_1[x \mapsto a][\Gamma' \mapsto \bar{a}'] \approx t_2[x \mapsto a][\Gamma' \mapsto \bar{a}']$ for all closed terms $\bar{a}' : \Gamma'$ by assumption. By the induction hypothesis, we have $t_1[x \mapsto a] \approx t_2[x \mapsto a]$. Now define $f_1 = \lambda(x : T).\ t_1$ and $f_2 = \lambda(x : T).\ t_2$. Then $f_1\ a \approx t_1[x \mapsto a] \approx t_2[x \mapsto a] \approx f_2\ a$ for all closed terms $a : T$, so $f_1 \approx f_2$ by the second rule. By the third rule, we have $t_1 \approx f_1\ x \approx f_2\ x \approx t_2$, as we wanted to prove.

Now suppose $t_1 \approx t_2$ and let σ be any substitution, then we will prove that $t_1\sigma \approx t_2\sigma$ by induction on the derivation of $t_1 \approx t_2$:

- If $\Gamma \vdash t_1 = t_2 : T$, then we have $\Gamma\sigma \vdash t_1\sigma = t_2\sigma : T\sigma$, hence $t_1\sigma \approx t_2\sigma$.
- If $t_1\ a \approx t_2\ a$ for all closed terms a, then we have $(t_1\ a)\sigma \approx (t_2\ a)\sigma$ by the induction hypothesis. Because a is closed, this implies $(t_1\sigma)\ a \approx (t_2\sigma)\ a$. This holds for any closed a, hence $t_1\sigma \approx t_2\sigma$.
- If $t_1 = f_1\ s_1$ and $t_2 = f_2\ s_2$ for $f_1 \approx f_2$ and $s_1 \approx s_2$, then we have $f_1\sigma \approx f_2\sigma$ and $s_1\sigma \approx s_2\sigma$ by the induction hypothesis. Hence we have $t_1\sigma = (f_1\sigma)\ (s_1\sigma) \approx (f_2\sigma)\ (s_2\sigma) = t_2\sigma$.

If we take $\sigma = [\Gamma \mapsto \bar{a}]$ in particular, then we get $t_1[\Gamma \mapsto \bar{a}] \approx t_2[\Gamma \mapsto \bar{a}]$, as we wanted to prove. □

Proof of theorem 8.2. Let P be the set of patterns in the definition of f. Because the clauses of f satisfy the requirements of 6.1, there exists a covering O such that for each $\bar{p} \in O$, there exists a $\bar{q} \in P$ such that $\bar{q} \supseteq \bar{p}$. In other words, for all $\bar{p} \in O$ there exists a clause $f\ \bar{q} = t$ for f and a substitution σ such that $\bar{q}\sigma = \bar{p}$. The function f' will be defined by the clauses $f'\ \bar{p} = t[f \mapsto f']\sigma$ for all $\bar{p} \in O$. We check that this is a valid definition:

- The set of patterns O is a covering, hence the patterns are valid and complete.
- The arguments of all recursive calls $f\ \bar{s}$ in the right-hand side of a clause $f\ \bar{q} = t$ satisfy $\bar{s} \prec \lceil \bar{q} \rceil$. This implies $\bar{s}\sigma \prec \lceil \bar{q} \rceil\sigma = \lceil \bar{p} \rceil$ by lemma 4.12. This implies that the definition of f' is terminating by theorem 4.11.
- The patterns in O do not overlap, hence the definition of f' is confluent.

Now we will prove that $f \approx f'$. By definition of $\approx$, it is sufficient to prove that $f\ \bar{a} \approx f'\ \bar{a}$ for all closed $\bar{a} : \Delta$. We will proceed by well-founded induction on the

structural order $\prec$. By completeness of f', there exists a clause $f'\ \bar{p} = s$ such that $\textsc{match}(\bar{p}, \bar{a}) \Rightarrow \tau$. This gives us that $\bar{a} = \lceil \bar{p} \rceil \tau$ and $f'\ \bar{a} \longrightarrow s\tau$. By construction of f', the clause $f'\ \bar{p} = s$ is of the form $f'\ \bar{q}\sigma = t[f \mapsto f']\sigma$ where $f\ \bar{q} = t$ is a clause of f. This implies that $\bar{a} = \lceil \bar{p} \rceil \tau = \lceil \bar{q} \rceil \sigma\tau$ and hence $f\ \bar{a} \longrightarrow t\sigma\tau = s[f' \mapsto f]\tau$.

We claim that $f\ \bar{u} \approx f'\ \bar{u}$ for each recursive call $f'\ \bar{u}$ in $s\tau$. By lemma 8.3, it is sufficient to prove that $f\ (\bar{u}[\Gamma \mapsto \bar{b}]) \approx f'\ (\bar{u}[\Gamma \mapsto \bar{b}])$ where Γ contains all free variables of $\bar{u}$ and $\bar{b} : \Gamma$ is closed. By the criterion for termination and lemma 4.12, we have that $\bar{u} \prec \bar{a}$, hence $\bar{u}[\Gamma \mapsto \bar{b}] \prec \bar{a}[\Gamma \mapsto \bar{b}] = \bar{a}$ by lemma 4.12 and the fact that $\bar{a}$ is closed. So by the induction hypothesis, we have $f\ (\bar{u}[\Gamma \mapsto \bar{b}]) \approx f'\ (\bar{u}[\Gamma \mapsto \bar{b}])$.

We now have $f\ \bar{u} \approx f'\ \bar{u}$ for each recursive call in $s\tau$. By applying the definition of $\approx$ and lemma 8.3 repeatedly, we get that $s\tau \approx s[f' \mapsto f]\tau$. So we have that $f\ \bar{a} \approx s[f' \mapsto f]\tau \approx s\tau \approx f'\ \bar{a}$ for all closed $\bar{a}$, which implies $f \approx f'$, as we wanted to prove. $\square$

Chapter 9

Conclusion and future work

The main goal of this thesis is to make dependent pattern matching more intuitively usable for specialists and non-specialists alike. We try to do this by fixing some discrepancies between the definitions given by the user and the internal representation of these definitions. To do this, we extend the semantics of pattern matching to sets of patterns that do not necessarily form a covering. In particular, we allow overlapping patterns. This allows us to give a lot of natural definitions that behave as we would expect them to.

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 type checker doesn't complain any more. This blends well with the typical interactive development of dependently typed programs in Agda.

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

There are some limits to our approach: 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 multiplication on natural numbers. 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. The current definition of type theory doesn't allow these kind of non-computational (undirected) definitional equalities. To allow such definitions, we would need to introduce the theory of AC (associative-commutative) rewriting [BP85] to type theory.

As with any modification to type theory, there is the question of soundness. Specifically, does our extended form of pattern matching allow us to give a closed term of type $\bot$? One would certainly hope that the answer is no. We think that our theorem 8.2 gives a step in the right direction, but it is too weak because it doesn't say much about the reduction rules involved. It is an interesting question what extra requirements we need to obtain a stronger equivalence with non-overlapping definitions.

Appendix A

Implementation

In chapter 6, we discussed the conceptual issues with the implementation of definitions with overlapping patterns, in specific the implementation of the confluence checker. We also gave some examples of definitions with overlapping patterns in chapter 7. In this appendix, we want to give a few additional remarks that are specific to the implementation of these features in Agda. First, we describe how definitions with overlapping patterns can be given; and second, we describe some issues with the implementation itself. As of this writing, a working version of Agda with our modifications has not yet been released. If you are interested in a copy, please contact the author at: `jesper.cockx@student.kuleuven.be`.

A.1 Usage

In order to maintain backwards compatibility with functions that use the first-match semantics, we added a new keyword `overlapping` to Agda that marks all clauses in the next definition to be interpreted as definitional equalities. Function definitions without this keyword are still translated to a case tree internally, according to the first-match semantics. Figure A.1 gives an example of how this keyword is used.

An `overlapping` keyword can be followed by a block containing multiple functions. The end of the block is determined by the indentation. Figure A.2 gives an example of such an `overlapping` block. This works in the same way as other keywords in Agda. If used in this way, an `overlapping` block doubles as a `mutual` block in Agda: all functions in the block can refer to each other, irrespective of their order. Confluence checking is also done for all definitions in the block at once. This could be useful if the confluence of two definitions each depends on the other definition.

In order to allow definitions such as `plus` in figure A.1, the built-in check for unreachable clauses has been disabled.

```
overlapping
  _+_ : ℕ → ℕ → ℕ
  zero    + n       = n
  (suc m) + n       = suc (m + n)
  m       + zero    = m
  m       + (suc n) = suc (m + n)
```

Figure A.1: Definition of addition with overlapping patterns in Agda. This definition is accepted by the confluence checker.

```
overlapping
  _∧_ : Bool → Bool → Bool
  x     ∧ false = false
  false ∧ y     = false
  x     ∧ true  = x
  true  ∧ y     = y

  _∨_ : Bool → Bool → Bool
  x ∨ true  = true
  true ∨ y  = true
  x ∨ false = x
  false ∨ y = y
```

Figure A.2: Overlapping block with multiple functions. Although it is not visible in this example, it is allowed that the functions in the block mutually refer to each other.

```
overlapping
  _*_ : ℕ → ℕ → ℕ
  zero  * n     = zero
  m     * zero  = zero
  suc m * n     = n + (m * n)
  m     * suc n = (m * n) + m
While checking overlap, @0 and @0 + (@1 * @0) were unequal.
```

Figure A.3: Definition of multiplication with overlapping patterns. This definition is rejected by the confluence checker. At the moment, something is preventing the right-hand sides in the error message from being displayed correctly.

Two new error messages have been added to Agda, corresponding to the two errors described in section 6.2. The first one occurs when the unification of two patterns succeeds positively, and the right-hand sides were not found to be equal after this unification. Figure A.3 gives an example where this error occurs.

The second error occurs when the unification of two patterns fails, violating the assumption of theorem 6.2. An example where this error occurs is given in figure A.4.

```
data _≟_ {A : Set} : A → A → Set where
  eq  : (x : A)   → x ≟ x
  neq : (x y : A) → x ≟ y

overlapping
  f : (x y : Bool) → x ≟ y → Bool
  f true  true p             = true
  f true  .y   (neq .true y) = y
  f false y    p             = y
Couldn't determine whether patterns overlap.
```

Figure A.4: An example where the unification algorithm fails. This could be improved by using a better unification algorithm.

A.2 Implementation issues

Here are some remarks on the problems we encountered when working with the source code of Agda.

- Adding a single field to the internal representation of a function in order to distinguish between old and new definitions caused quite a bit of trivial but non-local changes to the code. This is caused by the fact that the Agda source code often refers to the specific representation of a function definition, which could perhaps be done better by using getters for each field.
- The built-in unification algorithm of Agda didn't work for unifying patterns because it assumed that the De Bruijn-indices of the variables are disjoint. This is not the case for patterns, as each clause has its own scope. Hence we implemented a simple two-pass unification algorithm for patterns. In the first pass, the accessible parts of the patterns are unified, and in the second pass, pattern variables and inaccessible patterns are unified.
- The current Agda implementation didn't provide any substitution that directly substitutes a term for a variable without shifting the De Bruijn-indices. This kind of direct substitution was needed for the confluence checker, so we had to add it ourselves. The reason that this was needed, was that the variables in two different clauses can have the same De Bruijn-indices, without being 'the same' variable.

Hopefully, these remarks can also be useful for someone implementing a confluence checker for another language.

Bibliography

[AA02] A. Abel, and T. Altenkirch: *A predicative analysis of structural recursion.* Journal of Functional Programming 12.1, 2002. 1-41.

[BP85] L. Bachmair, and D. A. Plaisted, *Termination orderings for associative-commutative rewriting systems.* Journal of Symbolic Computation 1.4, 1985. 329-349.

[Coq92] T. Coquand, *Pattern matching with dependent types.* Proceedings of the 1992 workshop on types for proofs and programs, Båstad, Sweden, ed. B. Nordström et al., 1992. 66-79. Available at `www.cse.chalmers.se/research/group/logic/Types/proc92.ps` (accessed 3 June, 2013).

[Dyb94] P. Dybjer, *Inductive families.* Formal Aspects of Computing 6.4, 1994. 440-465.

[Gog94] H. Goguen, *A typed operational semantics for type theory.* PhD thesis, University of Edinburgh, 1994.

[Gon13] G. Gonthier, *Engineering mathematics: the odd order theorem proof.* Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Rome, Italy. New York: ACM, 2013. 1-2.

[GMM06] H. Goguen, C. McBride, and J. McKinna, *Eliminating dependent pattern matching.* Algebra, Meaning, and Computation, ed. K. Futatsugi et al. (Lecture Notes in Computer Science, 4060). Berlin/Heidelberg: Springer, 2006. 521-540.

[Gr91] A. Gräf, *Left-to-right tree pattern matching.* Proceedings of the 4th international conference on rewriting techniques and applications, Como, Italy, ed. R.V. Book (Lecture Notes in Computer Science, 488). Berlin/Heidelberg: Springer, 1991. 323-334.

[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. 81-92.

[Luo94] Z. Luo, *Computation and reasoning: a type theory for computer science* (International Series of Monographs on Computer Science, 11). New York/Oxford: Oxford University Press, Inc., 1994.

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

[MGM06] C. McBride, H. Goguen, and J. McKinna, *A few constructions on constructors.* Types for Proofs and Programs, ed. J.-C. Filliâtre et al. (Lecture Notes in Computer Science, 3839). Berlin/Heidelberg: Springer, 2006. 186-200.

[MM04] C. McBride, and J. McKinna, *The view from the left.* Journal of Functional Programming 14.1, 2004. 69-111.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's type theory* (International Series of Monographs on Computer Science, 7). New York/Oxford: Oxford University Press, Inc., 1990.

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

[Rij12] E. Rijke, *Homotopy type theory.* Master Thesis, Utrecht University, 2012.

[Soz10] M. Sozeau, *Equations: A dependent pattern-matching compiler.* Proceedings of the first international conference on interactive theorem proving, Edinburgh, UK, ed. M. Kaufmann et al. (Lecture Notes in Computer Science, 6172). Berlin/Heidelberg: Springer, 2010. 419-434.

[Tho99] S. Thompson, *Type theory and functional programming.* Computing Laboratory, University of Kent, 1999. Available at `http://www.cs.kent.ac.uk/people/staff/sjt/TTFP/` (accessed 3 June, 2013).

[VCW12] D. Vytiniotis, T. Coquand, and D. Wahlstedt, *Stop when you are Almost-Full.* Proceedings of the third international conference on interactive theorem proving, Princeton, NJ, USA, ed. L. Beringer and A. Felty (Lecture Notes in Computer Science, 7406). Berlin/Heidelberg: Springer, 2012. 250-265.

[Xi03] H. Xi, *Dependently typed pattern matching.* Journal of Universal Computer Science 9.8, 2003. 851-872.

Departement Wiskunde
Celestijnenlaan 200B
B-3001 LEUVEN (HEVERLEE), BELGIË
tel. + 32 16 32 70 15
fax + 32 16 32 79 98
secretariaat@wis.kuleuven.be
www.kuleuven.be