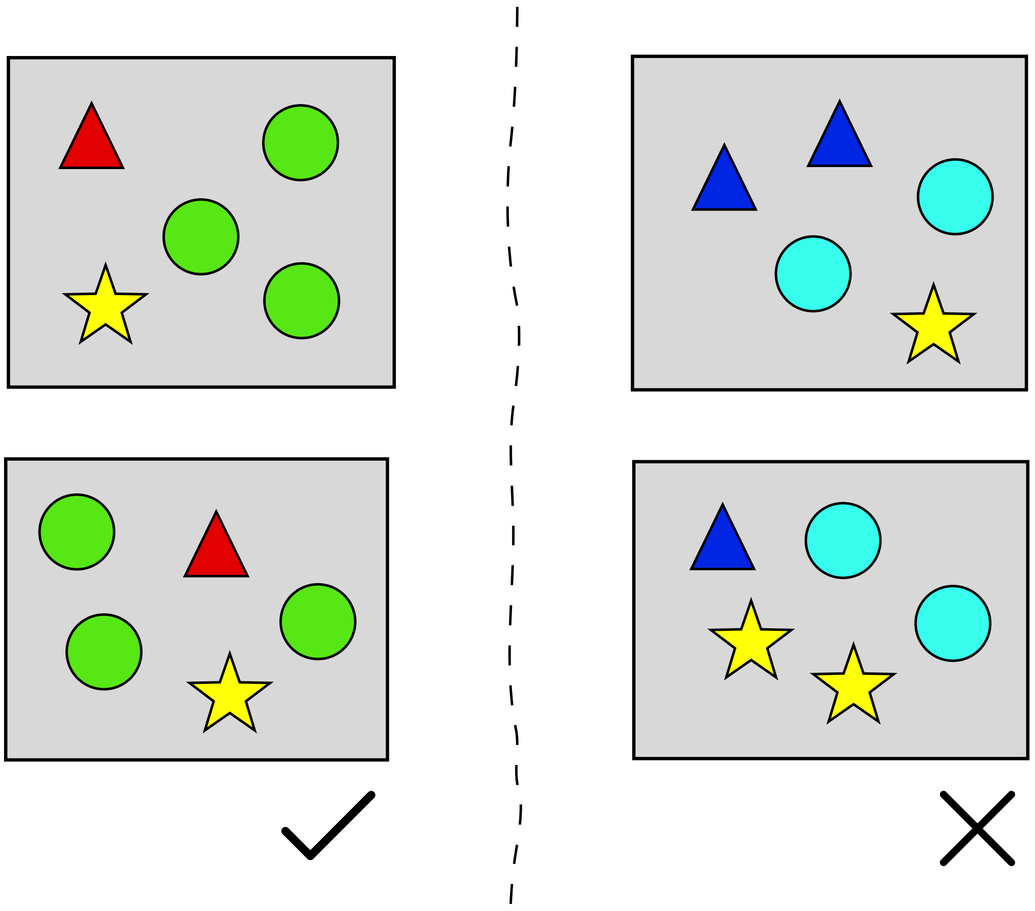
Dependent Type-Checking Modulo Associativity and Commutativity

Version of July 21, 2023



Lucas Holten

Dependent Type-Checking Modulo Associativity and Commutativity

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Lucas Holten
born in Rotterdam, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Cover picture: Comparing bags.

Dependent Type-Checking Modulo Associativity and Commutativity

Author: Lucas Holten
Student id: 4903692
Email: lhc.holten@gmail.com

Abstract

Writing software that follows its specification is important for many applications. One approach to guarantee this is formal verification in a dependently-typed programming language. Formal verification in these dependently-typed languages is based on proof writing. Sadly, while proofs are easy to check for computers, writing proofs can be tedious for developers. One particular proof component that currently requires developers attention in many systems, is associative and commutative (AC) reasoning.

We contribute an extension of dependent type-systems to fully automate AC reasoning. This alleviates developers from this task and allows them to concentrate on other proof components. Our approach works by modifying the conversion checker and doesn't compromise soundness or completeness. Furthermore, our approach reuses existing type-checking components, making it easier to implement. We also implemented our theory as an extension of the Agda type-checker. This allowed us to use this implementation to experiment with some example programs.

This thesis can help language designers decide if they want automatic AC reasoning in their language. For language users it can serve as inspiration on how to use such a type-system and finally for researchers we have ideas for future work.

Preface

I would like to thank everyone that supported me during the writing of this thesis. Special thanks to Jesper Cockx and Lucas Escot for their amazing feedback and for answering my questions. This work is definitely much better because of them and everyone else who gave me feedback.

Lucas Holten
Delft, the Netherlands
July 21, 2023

Contents

Preface iii

Contents v

1 Introduction 1
- 1.1 Research Questions 2
- 1.2 Contributions 2

2 Background 3
- 2.1 Agda Syntax 3
- 2.2 Equality 6
- 2.3 Rewrite Rules 7
- 2.4 Meta-Theory of Type Systems 7
- 2.5 Type-Checkers 8
- 2.6 Limitations of AC Rewriting 8

3 Theory 11
- 3.1 Lambda Pi Calculus 11
- 3.2 Lambda Pi modulo AC 16
- 3.3 Lambda Pi modulo AC and Rewrite Rules 18

4 Implementation 25
- 4.1 Agda Type Checking Overview 25
- 4.2 Scope of Implementation 26
- 4.3 Normalization 26
- 4.4 Conversion Checking 27
- 4.5 Injectivity Analysis 28

5 Usage Examples 31
- 5.1 Sorting Algorithms 31
- 5.2 Bags 33
- 5.3 Effect Systems 35

6 Related work 37
- 6.1 Implemented in the Type System 37
- 6.2 Non-Dependent Type Systems 37
- 6.3 Implemented with Macros 37

7 Discussion 39

7.1 Future work 40

Bibliography **41**

Acronyms **45**

A Improved Injectivity Analysis **47**

B Green Slime **49**
B.1 With-Abstraction and Instance Arguments 50

C Example Code **51**
C.1 Sorting Algorithms 51
C.2 Bag of Nats 54
C.3 Effect Rows 55

Chapter 1

Introduction

Writing tests gives us confidence that our software is correct. Other techniques like fuzzing can improve this confidence further. We get most confidence from formal verification, because it removes any doubt that software follows its implementation (Leino 2010; Rondon, Kawaguci, and Jhala 2008). Writing a proof in a dependently-typed language (Martin-Löf 1975b) is one method of formal verification. While it is especially light on computation compared to some other methods, it shifts the burden to the writer of the proof (Bruijn 1994).

This is why automatic proof writing tools, such as tactics[1] and macros[2], are used to make proof writing faster. It is even better however, to remove the need for a proof entirely, and this is sometimes possible by extending the type theory. Research in this direction has resulted in 'rewrite rules', that allow automating some equational reasoning (Cockx 2019). While this is already extremely useful, rewrite rules can not be used to perfectly capture associative and commutative (AC) functions. This is problematic because AC functions such as `+`, `*`, `minimum`, `&&` and `union` are quite common.

Our goal is to make it possible and straightforward to extend existing dependent type systems with the theory of AC functions. The approach that we take is to first define a minimal calculus that handles AC functions automatically. This calculus is then extended with rewrite rules to study the interaction between AC functions and rewrite rules. We include detailed explanations of type-checking procedures and justification of why these procedures are sound and complete.

To test the effectiveness and difficulty of adding the new theory into an existing type-system, we test it on Agda. Agda is a dependently-typed functional programming language.[3,4] It is a good project to extend with new ideas because of its research focus, but the source code is quite large and difficult for new contributors to modify. Thus the modifications were kept to a minimum and existing procedures were reused where possible. Agda also has meta-variables and we discuss their interaction with AC functions. The final implementation can be used to write code like this:

```
{-# COMMASSOC +-comm +-assoc #-}

swap-length : (A : Set) -> (m n : Nat) -> Vec A (m + n) -> Vec A (n + m)
swap-length A m n xs = xs
```

This code implements a cast between two `Vec` types that are indexed by syntactically different expressions. The type checker needs to verify that these expressions are equal in value before it can allow the cast. Normally, Agda is unable to automatically verify this requirement, but it can do so with the `COMMASSOC` pragma.

[1]https://coq.inria.fr/refman/proof-engine/tactics.html

[2]https://agda.readthedocs.io/en/latest/language/reflection.html#macros

[3]https://wiki.portal.chalmers.se/agda/pmwiki.php

[4]An introduction to Agda is presented in Chapter 1

We use the new Agda extension to formally verify some sorting functions. This is done without writing any proofs and without helping the type-checker in any way other than marking the correct functions as AC and adding the right rewrite rules. We also implement a simple effect system. This effect system uses the `COMMASSOC` pragma to build effect rows that have unordered effects.

1.1 Research Questions

Our main research questions is: "Can user-defined AC equalities be effectively integrated into the conversion checker of an existing dependently-typed language?". To answer this question we will first answer these other sub-questions:

- How do we formalize a dependent type theory with AC functions? Chapter 3.
- What are the procedures required to type-check this theory? Chapter 3.
- What is the interaction of AC functions and meta-variables in Agda? Chapter 4.
- How can we make use of Agda with AC functions? Chapter 5.

1.2 Contributions

Our contributions are mostly of interest to language designer who consider adding AC functions to their type theory. Others who might be interested are those who want to use a language with AC functions and want to know what the limitations are and how to build around them. This is a selection of our contributions:

- A minimal lambda calculus with AC functions. Section 3.2.
- Discussion of the required algorithms to type-check this calculus. Section 3.2.
- A fragment of rewrite rules on AC functions that is easy to execute. Section 3.3.2.
- An algorithm to check confluence of our rewrite rule fragment. Section 3.3.5.
- A formally verified sorting algorithm without proofs. Section 5.1.
- An AC Bag data-structure for natural numbers. Section 5.2.
- A proof of concept effect system with row-polymorphism. Section 5.3.
- An improved injectivity analysis procedure. Appendix A.

The source code for our Agda implementation of defintional AC functions is available on GitHub: `https://github.com/LHolten/agda-commassoc`.

Chapter 2

Background

Before we can investigate how to extend a dependent type system with AC rules, we first have to investigate how existing dependent type systems without AC rules work. This chapter gives an overview of dependent type systems, rewrite rules and meta-theory. Finally we explain why rewrite rules are not enough for AC functions.

Dependent types make a type system much more expressive. This is because dependent types are allowed to depend on run-time values (variables). This means, for example, that the output type of a function can depend on its input. An example is $(x : \texttt{Nat}) \rightarrow \texttt{Vec Nat}\ x$, which is the type of a function that takes a natural number as argument and returns a list of that many elements.

One use of dependent types is to express guarantees of algorithms as types. The type system is then able to check a proof that the code follows the specification. This makes projects such as CompCert C[1] possible. Dependent types also make a programming language useful as a proof assistants; the expressive type system allows expressing theorems as types and proofs as programs.[2] An example of a big proof written in a proof assistant is the Odd Order Theorem (Gonthier et al. 2013).

These seemingly different uses turn out to be closely related. The Curry-Howard correspondence states that programs and proofs are the same (Curry 1934). A dependently-typed language just has a type system powerful enough to express interesting propositions.

2.1 Agda Syntax

This section serves as an introduction to the Agda syntax. The intended purpose is to give an idea of how a dependently-typed language is used and the Agda syntax will be used throughout the rest of this thesis. More detailed description of Agda can be found in the Agda language reference[3]. Agda is a dependently-typed functional programming language that looks similar to Haskell. You can use it to write programs like the following:

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat

_+_ : Nat → Nat → Nat
zero + y = y
(suc x) + y = suc (x + y)
```

[1]CompCert C is a formally verified C compiler https://compcert.org/

[2]Also see 'Why dependent types matter' by Altenkirch, McBride, and McKinna (2005)

[3]`https://agda.readthedocs.io/en/latest/`

First we use the data syntax to declare a new inductive datatype called Nat. This Nat type represents our natural numbers. Nat has type Set, where Set is the type of types in Agda.[4] The Nat datatype is declared to have two constructors. The first constructor zero has no arguments and constructs a Nat. The second constructor suc has one argument of type Nat and also produces a Nat. Writing down what each constructor produces is redundant here, but it will be necessary for indexed data types such as Vec.

Next we define a function called _+_. Using underscores in a function name like this makes the function an infix operator. This allows us to write the + operator between its arguments. The function is defined through pattern matching. The first argument is split into the possible constructors and in each case we give the result.

Now let us look at some more complicated data types:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

We declare a new datatype constructor List, it is parameterized by a basic type (A : Set). Parameters like this are required to be given a name. The two constructors have the parameter name A in scope and need to produce a value of type List A.

The dependent Vec datatype however, while also parameterized by (A : Set), has an additional index. This is indicated by the fact that the sort is now Nat → Set instead of Set. The fact that the index is a value Nat instead of another type Set is what makes Vec a dependent type. The index does not have to be given a name, and is also not in scope for the constructors. Now every constructor is allowed to choose the value of the index. For example, the Vec.[] constructor produces a value of type Vec A zero, indicating that it has zero length. The other constructor Vec._::_ takes a vector of any length and produces a vector that is one element longer.

Note that the second constructor of Vec has a named argument {n : Nat}. The value of this argument is in scope for the rest of the type signature. This is another example of a dependent type. Arguments that use curly braces like this are implicit arguments. This means that they do not need to be written down while pattern matching or when calling the function.

We can sum both List Nat and Vec Nat n types like this:

```
sum : List Nat → Nat
sum [] = zero
sum (x :: xs) = x + sum xs

sumv : {n : Nat} → Vec Nat n → Nat
sumv [] = zero
sumv (x :: xs) = x + sumv xs
```

We need to use {n : Nat} to make the sumv function work for vectors of any length. To explain the benefits of indexed data types and dependent functions, here is the head function:

```
data Maybe (A : Set) : Set where
  just : A → Maybe A
  nothing : Maybe A
```

[4]Set is only the first universe level.

```
head : {A : Set} → List A → Maybe A
head [] = nothing
head (x :: xs) = just x

headv : {A : Set} {n : Nat} → Vec A (suc n) → A
headv (x :: xs) = x
```

The `head` function takes the first element of a list. However, because the list might be empty, it can not guarantee that there is a first element. It thus has to return a `Maybe A`.

The function `headv` indexes the `Vec` datatype with a length of `suc n` to require a vector of length at least one. This means that it is unnecessary to consider the empty vector. The type-checker guarantees that there is always a first element, thus the function can returns just `A`.

```
_++_ : {A : Set} {n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Another example that shows the usefulness of dependent types is vector concatenation (`_++_`). To understand why this example type-checks it is necessary to look at each branch in depth:

Looking at the first branch, Agda sees that the first argument is the empty vector and thus it knows that `n = 0`. The length of the second argument is `m` and the result should have length `n + m = 0 + m`. We have defined addition with a rule `0 + y = y`, so the required length computes to `0 + m = m`. The length of the value `ys` that we return is also `m`, so this branch type-checks.

The second branch is similar, the length of the first argument is `n = suc k`, where `k` is the length of `xs`. The length of the second argument is `m`. The required result length is `n + m = (suc k) + m`, we have defined addition with a rule `(suc x) + y = suc (x + y)`, so the required length is `(suc k) + m = suc (k + m)`. Indeed, if we calculate the length of the actual returned value `x :: (xs ++ ys)`, we will see that it is the same. Thus the second branch also type-checks.

Notice that the required computations line up perfectly with the computation rules of addition. This is the only reason that this works, if we define addition to split on the second argument we get this error:

```
m != zero + m of type Nat
when checking that the expression ys has type Vec A (zero + m)
```

If we change addition back to split on the first argument and define concatenation to return `Vec A (m + n)` instead of `Vec A (n + m)`, we get a similar looking error:

```
m != m + zero of type Nat
when checking that the expression ys has type Vec A (m + zero)
```

Finally if we do both modifications we get this error:

```
m != zero of type Nat
when checking that the expression ys has type Vec A (m + zero)
```

For this last error, the injectivity analysis makes an assumption based on our implementation, which makes the error more confusing. In general, these valid modifications are not accepted, because their types can not be checked automatically.

2.2 Equality

In order to check your work, a type-checker needs to check that each term has the expected type. For a dependent type system this means checking the equality of terms depending on variables. Thus a dependent type theory requires rules about which terms with variables can be considered equal. Deciding all equal terms is unfeasible, because it would require proving arbitrary theorems. Instead it is common to carefully extend the set of equalities, that are checked automatically, with a decidable set of rules.

Definitional Equality

The equalities that are checked automatically are called 'definitional'. When this works we get the frictionless typing that can be seen in the concatenation example from section 2.1. Two definitional equalities that are relevant to this thesis are α-conversion and β-reduction. α-conversion is definitional in every dependent lambda calculus:

$$\lambda x.M = \lambda y.M[x := y]$$

This makes it so that the names of variables do not matter. An example would be:

$$\lambda x.xz = \lambda y.yz$$

β-reduction is also definitional in most[5] dependent type theories:

$$(\lambda x.M)V = M[x := V]$$

The left side of the equivalence is called a β-redex and this equality is what makes lambda expressions compute inside types. An example reduction would be:

$$(\lambda x.xz)f = fz$$

Next to α-conversion and β-reduction, Agda also has η-expansion and δ/ι-reduction. These are not relevant to this thesis and we will thus not describe them.

Propositional Equality

Where definitional equalities only include those that the type-checker can prove automatically, propositional equalities do not have such restriction. Propositional equalities include all equalities that are provable. To build these proofs we use the principle of propositions as types. The dependent equality type $a \equiv b$ represents the proposition that a is equal to b and any value of this type is proof that the equality holds. In Agda we can define the equality type like this:

```
data _≡_ {A : Set} : (x y : A) -> Set where
  refl : {v : A} -> v ≡ v
```

The constructor of this type, `refl`, asserts that the two arguments of the type are definitionally equal. An instance of an equality type can then be destructed to make its arguments definitionally equal again.[6] This allows building proofs based on congruence, induction and case analysis. The example below shows how to prove $x + 0 \equiv x$ for all x:

```
cong : ∀ {A B : Set} (P : A → B) {x y : A}
  → x ≡ y → P x ≡ P y
```

[5]Zombie does not have definitional β-reduction, instead it has definitional congruence closures (Sjöberg and Weirich 2015).

[6]This does not always work as described in appendix B.

```
cong _ refl = refl

x+0≡x : (x : Nat) → (x + zero) ≡ x
x+0≡x zero = refl
x+0≡x (suc x) = cong suc (x+0≡x x)
```

`x+0≡x` is a single symbol which we use as the name of the propositional proof. Our proof is by induction over the first argument. The base case is resolved with `refl` and the induction step is proven with congruence and the induction hypothesis `x+0≡x x`. This proof can then be used to type cast between the relevant types:

```
cast : ∀ {A : Set} (P : A → Set)
  {x y : A} → x ≡ y → P x → P y
cast _ refl x = x

cast-vec : {A : Set} {x : Nat} → Vec A (x + zero) → Vec A x
cast-vec {A} {x} v = cast (Vec A) (x+0≡x x) v
```

2.3 Rewrite Rules

Even with the definitional equalities from section 2.2, there are still a lot of propositional proofs that need to invoked manually. This is why Agda has rewrite rules. Rewrite rules allow invoking propositional proofs automatically, effectively making them definitional.

For our running example, we might want Agda to be able to see that `Vec Bool (a + 0)` is the same type as `Vec Bool a`. Agda does not know about this equality by default, because addition is defined to pattern match on the first argument like this: `0 + x = x`. We can add the propositional proof `x+0≡x` from section 2.2 as a rewrite rule to the theory with `{-# REWRITE x+0≡x #-}`. With the rewrite rule added to the theory, Agda can now see that the two types are equal, without manual invocation of the proof. Thus the `cast-vec` example from section 2.2 can be written as:

```
{-# REWRITE x+0≡x #-}

cast-vec' : {A : Set} {x : Nat} → Vec A (x + zero) → Vec A x
cast-vec' v = v
```

2.4 Meta-Theory of Type Systems

Meta-theory studies the properties of type theories themselves. Some of these properties are important to make a type theory useful in practice. The ones that we will discuss are consistency and subject reduction.

Consistency

Consistency means that it is impossible to prove contradictions. The Curry-Howard correspondence says that a proposition is provable if and only if the corresponding type is inhabited. A type is inhabited if there exists an expression with that type. Without an uninhabited type, every proposition would be provable. The existence of an uninhabited type is thus required for consistency. However, the existence of an uninhabited type is also sufficient for consistency. This is because a provable contradiction would prove any other proposition and leave no uninhabited types.

The required uninhabited type is often denoted with the symbol $\bot$, called bottom. $\bot$ can be used to write negations like $\neg P = P \to \bot$. When algebraic data types are available, it is possible to define $\bot$ as the data type without any constructors. A function that returns a value of any type disproves the existence of an uninhabited type and thus also makes a type system inconsistent. The signature of such a function would be $(T : \textbf{Type}) \to T$. A function type like this should thus also be uninhabited in a consistent type theory. This means that a consistent type theory can not have general recursion or infinite loops, because that would allow implementing a function of type $(T : \textbf{Type}) \to T$.

If we make all propositional equalities definitional, then we get an extensional type theory. The reason why we do not use an extensional type theory is because type-checking extensional type theory is undecidable. However, an important result is that extensional type theory is consistent (Martin-Löf 1975a). This means that if we are able to prove an equality in a consistent type theory, then we can make that equality definitional without losing consistency.

Subject Reduction

Subject reduction means that every reduction step preserves the type definitionally. This means that if a has some type T and a reduces to some value b, then b also has type T. Without subject reduction, it would not be possible to guarantee that after computation, the term still type-checks against its original type. We have subject reduction if and only if we have well-typedness and 'product compatibility'. Of which product compatibility can be proven through the stronger property of confluence. (Saillard 2015). Well-typedness and confluence will be discussed in chapter 3.

If one wants to preserve subject reduction while adding rewrite rules, it is sometimes necessary to make additional propositional equalities definitional. This can require a combination or rewrite rules and the definitional AC functions as described in this thesis.

2.5 Type-Checkers

Manually verifying the correctness of proofs written in a dependent language would be error prone and extremely time consuming. Thus, for practical use, a good type-checker is vital. This type-checker needs to be sound and complete:

- Soundness means that the type-checker never accepts a program that is not correct according to the type system. This ensures that we can trust that if a program is verified by a type-checker, then it is also correct according to the type system.
- Completeness means that the type-checker always accepts a program that is correct according to the type system. This ensures that if we write a correct program, then it will always be accepted by the type-checker.

Note that this only requires the type-checker to be a semi-decision procedure. It might still run forever for incorrect programs and proofs. This is less of a problem, because the program or proof can be fixed if there is suspicion that the type-checker might be stuck.

2.6 Limitations of AC Rewriting

Applying rewrite rules exhaustively is preferably terminating for well-typed terms. Otherwise the type checker can not guarantee completeness by just exhaustively applying rewrite rules. Since commutativity `(a b : Nat) → a + b ≡ b + a` is symmetric, adding it as a rewrite rule would make the rewrite system non-terminating. It would thus make the type-checker non-terminating when it tries to apply this rule exhaustively.

Associativity `(x y z : Nat) → (x + y) + z ≡ x + (y + z)` is also not perfectly suitable as a rewrite rule. Although it does not make the rewrite system non-terminating, there are combinations with other associative properties that can not be expressed with a finite number of rules (Peterson and Stickel 1981).

Chapter 3

Theory

The previous section gave some background knowledge and motivated the need for a type-checker with definitional AC functions. In order to implement such a type-checker, we first need to establish the core theory that it should abide by. In this chapter we will thus present a minimal type theory, enabling us to declare specific functions as definitionally AC.

Lambda Pi ($\lambda\Pi$) (Harper, Honsell, and Plotkin 1993) is used as a foundation for our calculus, because of its simplicity and its natural support for extensions. To extend this calculus with AC functions, we add a few inference rules to our $\lambda\Pi$ foundation. Finally, we add rewrite rules to the calculus in order to approximate existing, practical type theories.

The presentation of inference rules is targeted at language designers, who want to implement the calculus or extend another calculus with the theory of AC functions.

3.1 Lambda Pi Calculus

The $\lambda\Pi$ calculus, also known as the Edinburgh Logic Framework or LF, is a minimalist dependently typed lambda calculus. As an example of its generality, it is possible to represent first- and higher-order logic in $\lambda\Pi$ (Harper, Honsell, and Plotkin 1993). The calculus works with so called 'constants', that encode the relevant theory in the context. These constants can be thought of as axioms or postulates. The focus on encoding theory in the context is what makes $\lambda\Pi$ a good foundation to build an extension that support the theory of AC functions. In this section we will thus give an overview of $\lambda\Pi$ with its typing judgements.

objects		types		kinds		
10	:	$\texttt{nat}$	:	**Type**	:	**Kind**
$\lambda x.\, x$	:	$\Pi x : \texttt{nat}.\, \texttt{nat}$	:	**Type**	:	**Kind**
		$\lambda x.\, \texttt{bitvec}\ x$	:	$\Pi x : \texttt{nat}.$ **Type**	:	**Kind**

Terms in $\lambda\Pi$ can be separated into three distinct groups: objects, which are assigned a type, types which themselves are assigned a kind and finally kinds. Object level terms are the only ones that can be used as arguments to functions.

For the notation of $\lambda\Pi$, we use a mix of the conventions from Boespflug, Carbonneaux, and Hermant (2012). Our notation has a context, but does not have type annotations in lambda expressions. The grammar consists of a single production rule for terms:

$$K, A, B, M, N ::= x \mid \textbf{Type} \mid \textbf{Kind} \mid \Pi x : A.\, K \mid \lambda x.\, B \mid M\ N$$

In this calculus there is an infinite number of lowercase names x, y, z. Lowercase names are used for both variables bound in lambda terms $\lambda x.x$ and for constants that are given a type in the context. As can be seen later in the inference rules, variables become constants in the body of the lambda term where they are defined. α-conversion is only relevant for notation, so it is assumed to be automatic in all of the inference rules. An example of α-conversion is show in section 2.2.

- The notation $\Pi x : A.\, K$ is used for dependent function types, the same would be written $(x : A) \to K$ in Agda. We also refer to Π as the dependent product.
- The notation $\lambda x.\, B$ is used for lambda expressions. Here the type of the argument is not written. The type of the argument can be inferred with bidirectional type checking as described in 3.1.1.
- The notation $M\ N$ is used for function application.
- The notation $A \to K$ is used as a shorthand notation for $\Pi x : A.\, K$ when x is not used in K.

As an example, the type of concatenation of bit-vectors indexed by their length could be written as:

$$\begin{aligned}
&\Pi m : \texttt{nat}.\, \Pi n : \texttt{nat}.\\
&\quad \texttt{bitvec}\ m \to\\
&\quad \texttt{bitvec}\ n \to\\
&\quad \texttt{bitvec}\ (\texttt{plus}\ m\ n)
\end{aligned}$$

The sub-terms `nat`, `bitvec` and `plus` would be either variables or constants. Note that we can not write `vec bool`, because all arguments must be objects and `bool` is a type.

3.1.1 Bidirectional Type Checking

Bidirectional typing is split into two procedures. This separation is necessary because there are two cases that can occur during type checking. The first one is that we know what the type of the terms should be, but the term does not necessarily have a unique type. The second one being that we do not know what the type of the term should be, but we know that the term has a unique type.

These two procedures are called 'checks-against' for the case where we know what the type should be and 'synthesizes-to' for the case where we know that the term has a unique type. For both procedures we have a typing judgement with inference rules that specifies what types the procedure is allowed to accept or produce. The inference rules are such that there is always only one that can be used. This means that it is unnecessary to do any back-tracking and makes implementation straightforward.

- The 'checks-against' judgement $\Gamma \vdash M \Leftarrow A$ says that in context Γ, the term M has type A. Checks-against works on the assumption that context Γ is well-formed and term A is well-typed in Γ.
- The 'synthesizes-to' judgement $\Gamma \vdash M \Rightarrow A$ says that in context Γ, the term M has a unique type A. Synthesizes-to works on the assumption that context Γ is well-formed.

The inference rules of the above two type judgements depend on two extra relations. The following judgements represent these relations between terms:

- The 'reduction' relation $M \longrightarrow_{\beta\Gamma} N$ says that with the rewrite rules in Γ and β-reduction, the term M reduces to N in one step. Reduction works on the assumption that Γ is well-formed and M is well-typed in Γ.
- The 'equivalence' relation $M \sim_{\Gamma} N$ says that with the AC rules in Γ, the terms M and N are equal in one step. Equivalence works on the assumption that Γ is well-formed and M and N are well-typed in Γ.

The previous judgements work on the assumption that context is well-formed and/or some terms are well-typed. These properties are again judgements of which the definition is a set of inference rules.

- The 'well-formed' judgement $\vdash \Gamma$ says that context Γ is well-formed.
- The 'well-typed' judgement $\Gamma \vdash A$ says that term A is well-typed in context Γ. Well-typedness works on the assumption that Γ is well-formed.

A type-checker implementation would first check that the context is well-formed, then it would check that the type term is well-typed and finally it would use the checks-against procedure to check the object term against the type term.

3.1.2 Inference Rules

Each judgement has a set of inference rules. An inference rule looks like this:

$$\frac{premise1 \qquad ... \qquad premiseN}{conclusion}\ \text{RULE-NAME}$$

These rules can be used to implement an algorithm that tries to find a proof for a judgement. This can be as straightforwards as matching the conclusion of each inference rule with the judgement that needs to be proven. Then, if only one inference rule matches, the type-checker tries to find a proof for each premise of that rule.

Rules for Well-Formed Contexts

$$\frac{}{\vdash \langle\rangle}\ \text{EMPTY} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A \Rightarrow s \qquad x \notin dom(\Gamma)}{\vdash \Gamma, x : A}\ \text{CONSTANT} \quad s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

The context contains all the constants. These ought to be considered as either postulates in your theory, or previous declarations that are opaque to the typing judgement of interest. The context is what defines the theory that we are working with. It is the responsibility of the user to only type-check in a consistent context as described in section 2.4 . In order to guarantee subject reduction, it is enough to check that the constants are well-typed (Saillard 2015).

Rules for Well-Typed Terms

$$\frac{}{\Gamma \vdash \mathbf{Kind}}\ \text{KIND} \qquad \frac{\Gamma \vdash A \Rightarrow s}{\Gamma \vdash A}\ \text{TYPED-TERM} \quad s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

The only requirement for the well-typed judgement is that the context is well-formed. For a term to be well-typed it needs to synthesize-to **Type** or **Kind** or it needs to be **Kind** itself. This is equivalent to requiring a chain of types up to **Kind**, but it does not allow function terms, because those would have a product type.

Rules for 'checks-against'

$$\frac{\Gamma, x : A \vdash B \Leftarrow K \qquad C \longrightarrow^{*}_{\beta\Gamma} (\Pi x : A.K) \qquad x \notin dom(\Gamma)}{\Gamma \vdash (\lambda x.B) \Leftarrow C}\ \text{ABSTRACTION}$$

$$\frac{\Gamma \vdash A \Rightarrow C \qquad C \longrightarrow^{*}_{\beta\Gamma} K \qquad C' \longrightarrow^{*}_{\beta\Gamma} K' \qquad K \sim^{*}_{\Gamma} K'}{\Gamma \vdash A \Leftarrow C'}\ \text{CONVERSION}$$

All terms can be checked against a type. The only requirement is that the context is well-formed and that the type is well-typed.

- The ABSTRACTION rule is the first that uses the reduction relation $A \longrightarrow_{\beta\Gamma} B$. The reduction relation will be described later. In the case of the 'abstraction' rule it is used to extract the argument type. So that it is possible to type-check the lambda expression.

  The ABSTRACTION rule also needs to check that $x \notin dom(\Gamma)$ in order to guarantee that the context is well-typed when checking the function body. Note that we assume automatic α-conversion, so it is always possible to choose a fresh variable name to make this condition pass.

- The CONVERSION rule can be used to check terms that synthesize to a unique type. The CONVERSION rule will first reduce both types and then use the equivalence relation $A \sim_{\Gamma} B$ to check if the reduced forms of the types are equal.

Remark. Both the reduction relation and the equivalence relation are used with a star ($*$). This means that the relation can be applied multiple times, which is called the reflexive and transitive closure.

Rules for 'synthesizes-to'

$$\frac{}{\Gamma \vdash \textbf{Type} \Rightarrow \textbf{Kind}}\ \text{SORT} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \text{VARIABLE}$$

$$\frac{\Gamma \vdash A \Rightarrow \textbf{Type} \qquad \Gamma, x : A \vdash K \Rightarrow s}{\Gamma \vdash (\Pi x : A.K) \Rightarrow s}\ \text{PRODUCT} \qquad s \in \{\textbf{Type}, \textbf{Kind}\}$$

$$\frac{\Gamma \vdash A \Rightarrow C \qquad C \longrightarrow^{*}_{\beta\Gamma} (\Pi x : B.K) \qquad \Gamma \vdash M \Leftarrow B}{\Gamma \vdash AM \Rightarrow K[x := M]}\ \text{APPLICATION}$$

The 'synthesizes-to' judgement can be used to get the type for any neutral term. Neutral terms are those that will not create a β-redex when used as the first term in an application. Our definition of neutral terms also includes terms that do not have a product type and can thus not occur as the first term in an application at all. Conveniently this includes all terms that are the type of some other term. This is why we can use the 'syntesizes-to' judgement to check well-typedness and to check the types in a product term.

This judgement gives a unique type to these terms and only requires the context to be well-formed. It guarantees that the term has the specific type and that the type itself is well-typed. The APPLICATION rule requires that the left side synthesizes to a type that reduces to a Π term. This reduction is required, because the argument type is used to check the argument and the return type is used as the type of the application.

Rules for the Reduction Relation

PRODUCT-LEFT
$$\frac{A \longrightarrow_{\beta\Gamma} A'}{(\Pi x : A.B) \longrightarrow_{\beta\Gamma} (\Pi x : A'.B)}$$

PRODUCT-RIGHT
$$\frac{B \longrightarrow_{\beta\Gamma} B'}{(\Pi x : A.B) \longrightarrow_{\beta\Gamma} (\Pi x : A.B')}$$

APPLICATION-LEFT
$$\frac{M \longrightarrow_{\beta\Gamma} M'}{MN \longrightarrow_{\beta\Gamma} M'N}$$

APPLICATION-RIGHT
$$\frac{N \longrightarrow_{\beta\Gamma} N'}{MN \longrightarrow_{\beta\Gamma} MN'}$$

BETA
$$\frac{}{(x.M)N \longrightarrow_{\beta\Gamma} M[x := N]}$$

ABSTRACTION
$$\frac{A \longrightarrow_{\beta\Gamma} A'}{(\lambda x.A) \longrightarrow_{\beta\Gamma} (\lambda x.A')}$$

The reduction relation is single step β-reduction, where β-reduction is applied to one β-redex at a time. The only requirement is that the term is type-checked before reduction. In that case β-reduction has been proven to be strongly normalizing in $\lambda\Pi$ (Harper, Honsell, and Plotkin 1993). So applying this relation exhaustively always gives the same normal form.

Rules for the Equivalence Relation

PRODUCT-LEFT
$$\frac{A \sim_{\Gamma} A'}{(\Pi x : A.B) \sim_{\Gamma} (\Pi x : A'.B)}$$

PRODUCT-RIGHT
$$\frac{B \sim_{\Gamma} B'}{(\Pi x : A.B) \sim_{\Gamma} (\Pi x : A.B')}$$

APPLICATION-LEFT
$$\frac{M \sim_{\Gamma} M'}{MN \sim_{\Gamma} M'N}$$

APPLICATION-RIGHT
$$\frac{N \sim_{\Gamma} N'}{MN \sim_{\Gamma} MN'}$$

ABSTRACTION
$$\frac{A \sim_{\Gamma} A'}{(\lambda x.A) \sim_{\Gamma} (\lambda x.A')}$$

For now the reflexive and transitive closure of the equivalence relation is just syntactic equality. Checking if two terms are equivalent is decidable, the algorithm only needs to recursively check syntactic equality.

3.1.3 Conversion Checking

The conversion checking procedure corresponds precisely to the 'conversion' rule from the 'checks-against' relation. To reach its goal it uses a combination of the reduction rules and equivalence rules. The general method is quite straightforward: Start out by applying the reduction rules exhaustively. This results in a term that is normalized. Then check for equivalence using the rules from the equivalence relation.

The procedure is sound, because it uses steps that are justified by rules from the reduction and equivalence relation. The procedure is also complete, because $\lambda\Pi$ has been proven strongly normalizing. After normalizing, all that remains is to check for syntactic equality, which is also sound and complete.

Remark. Soundness here means that the conversion checker never accepts terms that are not definitionally equal. Completeness here means that the conversion checker always accepts terms that are definitionally equal. See section 2.5.

3.2 Lambda Pi modulo AC

In the previous section we have define the base for our calculus. In this section we extend the theory to create a minimal dependent type system where terms can be defined as associative and commutative.

Rules for Well-Formed Contexts

$$\textsc{commassoc}\quad \frac{\vdash \Gamma \qquad f : P_0 \to \cdots \to P_n \to (T \to T \to T) \in \Gamma}{\vdash \Gamma, \mathrm{AC}_n(f)}$$

Remark. We write $P_0 \to \cdots \to P_n \to T$ to indicate a function that first takes n parameters of some type and then returns a T.

We extend the definition of well-formed context to allow functions to be declared as associative and commutative with the commassoc rule. The only functions that we allow declaring AC are constants like f applied to some number n of parameters. There are two reasons why we do not allow specifying for which parameters a function is AC:

- General patterns would require general matching in order to check if a function is AC. Because functions can now be declared as definitionally AC, it would require AC matching. AC matching can be expensive so we would like to avoid it.
- When we add rewrite rules to our calculus, we would like to know which functions in the rewrite rule patterns are AC. Since pattern can contain free variables, this would not be possible with functions that are only declared to be AC for some parameters.

Definition 3.2.1. An AC function is an application of a constant f to n parameters where $\mathrm{AC}_n(f) \in \Gamma$. An AC function thus has type $T \to T \to T$ for some T and is definitionally associative and commutative.

Definition 3.2.2. An *AC term* is an application of an AC function to two arguments.

Definition 3.2.3. Two sub-terms are *interchangeable* if they can be swapped such that the resulting term is related to the original using the reflexive and transitive closure of the equivalence relation: $original \sim_\Gamma^* swapped$.

Definition 3.2.4. The *elements* of an AC term B are the sub-terms that are interchangeable with the last two arguments of B and that are themselves not applications of the same AC function as B.

Rules for the Equivalence Relation

$$\textsc{commutative}\quad \frac{\mathrm{AC}_n(f) \in \Gamma \qquad M = f\ A_0 \ldots A_n}{M\ X\ Y \sim_\Gamma M\ Y\ X} \qquad \textsc{associative-left}\quad \frac{\mathrm{AC}_n(f) \in \Gamma \qquad M = f\ A_0 \ldots A_n}{M\ X\ (M\ Y\ Z) \sim_\Gamma M\ (M\ X\ Y)\ Z}$$

$$\textsc{associative-right}\quad \frac{\mathrm{AC}_n(f) \in \Gamma \qquad M = f\ A_0 \ldots A_n}{M\ (M\ X\ Y)\ Z \sim_\Gamma M\ X\ (M\ Y\ Z)}$$

Remark. We use the notation $X = Y$ for syntactic equality between X and Y.

To make commutativity and associativity definitional, we extend the equivalence relation with the commutative, associative-left and associative-right rules. These are the only rules in the equivalence relation that actually do something, which means that we can decide equivalence by flattening and sorting the elements in each AC term. This is used in section 3.2.1 for a conversion checking procedure.

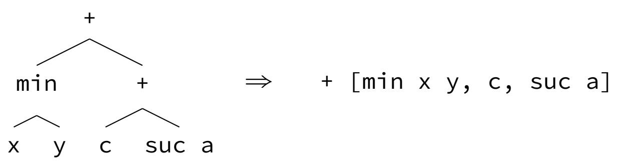


Figure 3.1: Flattening an AC-term using `+` function.

3.2.1 Conversion Checking Modulo AC

The conversion checking algorithm from section 3.1.3 for $\lambda\Pi$ without AC is not complete if the theory contains AC terms. The recursive equivalence check on arguments can miss solutions where the order of the elements is changed.

Instead we will now use the following algorithm:

1. Both terms are reduced completely using the reduction relation as before. This gives us normalized terms.

2. We will turn these normalized terms into canonical terms by flattening each AC term application and sorting the elements.

   Remark. `f [a, b, c]` is the notation of a flattened AC function `f` that is applied to some number of elements `a, b, c`.

   An application of an AC function can be flattened by comparing the function to the function in each argument and then taking the arguments of equal functions recursively. This gives us all elements as can be seen in figure 3.1

   Definition 3.2.5. *Sorting terms* is defined to be in any total order that is syntactic. This means that terms that are not syntactically equal will always have the same ordering between them.

   For example, the term `f a (f c b)` would be flattened into `f [a, c, b]` which is then sorted to become `f [a, b, c]`. This procedure is performed in a bottom up fashion, because sorting is only possible if the subterms are also in canonical form. Similarly it is necessary to canonicalize the function terms before flattening. Otherwise flattening can miss some definitionally equal functions.

3. When both terms are in canonical form, it is only necessary to check them for syntactic equality.

Lemma 3.2.1. The conversion checking algorithm is sound.

Proof. The algorithm is sound if every step it takes is justified by the theory. The first step of of applying reduction exhaustively is sound, because of the rules in the reduction relation. The second step of flattening the AC function applications and sorting their elements is also sound. This is justified because AC function symbols have definitional commutativity and associativity, which allows reordering their elements. If the canonical forms are equal, then the terms are also definitionally equal. Thus the algorithm is sound. □

Lemma 3.2.2. The conversion checking algorithm is complete.

Proof. The algorithm is complete if all definitionally equal terms convert successfully. Because β-reduction is strongly normalizing, we know that terms modulo β will have the same normal form. The most general structure that is associative and commutative is the multi-set. We know that sorted lists are a canonical form of multi-sets, so sorting the element list will make terms equal modulo AC syntactically equal. If the terms are definitionally equal, then the terms have equal canonical forms. Thus the algorithm is complete. □

3.3 Lambda Pi modulo AC and Rewrite Rules

Now that we have our minimal calculus with AC, it is useful to know how we can make it closer in expressiveness to full dependent programming languages like Agda. This gives insight in how AC interacts with the remaining features. The extension we chose to add to the core calculus is rewrite rules.

Definition 3.3.1. *Rewrite rules* are written using the notation $A \hookrightarrow B$. This says that terms matching the pattern A will rewrite to B.

Definition 3.3.2. *Free variables* are variables that are not bound by the context or any lambda or product constructor. To make the distinction easier, we will prefix free variables by an underscore such as `_1`.

Definition 3.3.3. We define *patterns* to be terms that can contain free variables.

Rewrite rules have been a common extension of $\lambda\Pi$ in previous research (Blanqui 2020; Boespflug, Carbonneaux, and Hermant 2012). This is because rewrite rules can simulate many desirable features from programming languages. For example they allow embedding any pure type system (Cousineau and Dowek 2007). Two other examples of features that are made possible with rewrite rules are presented below:

Type Polymorphism

One might want to encode a **Type** family, like $List$, that takes another **Type** as first argument. The problem is that a direct representation is not possible in $\lambda\Pi$, because functions only take objects as arguments. Instead we can define an embedding of types into objects, that we call a universe. The only thing needed then is a function that lifts these objects back to the type level. This function El is defined with rewrite rules. As an example we can have this theory:

$$\begin{aligned}
&Set : \textbf{Type}\\
&El : Set \to \textbf{Type}\\
&Nat : \textbf{Type}\\
&nat : Set\\
&El\ nat \hookrightarrow Nat\\
&List : Set \to \textbf{Type}
\end{aligned}$$

Here Set is our universe and it contains the embedding nat for the actual type Nat. With this we can express the type of `cons`:

$$cons : \Pi t : Set.\ El\ t \to List\ t \to List\ t$$

Pattern matching

Rewrite rules can also be used to emulate pattern matching by first building a case tree to get a set of non-overlapping patterns. This is necessary to make first-match pattern matching definitional. As an example, consider a function defined in Agda like this:

```
min : Nat -> Nat -> Nat
min zero y = zero
min x zero = zero
min (suc x) (suc y) = suc (min x y)
```

We could add it to the theory with rewrite rules like this:

$$\begin{aligned}
&Nat : \textbf{Type}\\
&zero : Nat\\
&suc : Nat \to Nat\\
&min : Nat \to Nat \to Nat\\
&min\ zero\ \_1 \hookrightarrow zero\\
&min\ (suc\ \_1)\ zero \hookrightarrow zero\\
&min\ (suc\ \_1)\ (suc\ \_2) \hookrightarrow suc\ (min\ \_1\ \_2)
\end{aligned}$$

Notice that $min\ (suc\ \_1)\ zero \hookrightarrow zero$ requires the first argument to be $suc\ \_1$ in order to give first match semantics. Alternatively, it is also possible to discard the first-match semantics of pattern matching. This idea is explored in the paper about 'Overlapping and Order-Independent Patterns' (Cockx, Piessens, and Devriese 2014).

3.3.1 Inference Rules

Now that we have seen some use cases, we take a look at the inference rules that are required to extend our calculus with rewrite rules.

Definition 3.3.4. A *substitution* σ maps free variables to terms. Applying a substitution to a pattern A is written $A[\sigma]$ and replaces free variables in A by the terms that they are mapped to.

Definition 3.3.5. If a pattern P *matches* a term A then there exists a substitution of the free variables σ that makes the pattern equal to the target, written as $P[\sigma] = A$.

Rules for Well-Formed Contexts

$$\frac{\vdash \Gamma \qquad \Gamma \vdash A[\sigma] \Leftarrow T \implies \Gamma \vdash B[\sigma] \Leftarrow T}{\vdash \Gamma, A \hookrightarrow B}\ \textsc{rewrite}$$

Just as with the COMMASSOC and CONSTANT rules, we can extend the theory by adding rewrite rules to the context using the REWRITE rule. To preserve subject reduction, we require well-typedness of the rewrite rules and confluence. Confluence checking will be described in section 3.3.5.

Well-typedness requires that every rewrite rule is type-preserving. Formally if $\Gamma \vdash A[\sigma] \Leftarrow T$ and $\Gamma \vdash A \hookrightarrow B$, then $\Gamma \vdash B[\sigma] \Leftarrow T$ for all Γ, A, B, T and σ. This is checked by the REWRITE inference rule for well-formed contexts. Note that A and B here are patterns, which means that they can contain free variables.

If we want the resulting theory to be consistent, then new rewrite rules also have to be consistent with the theory.

Rules for the Reduction Relation

$$\frac{M \sim^*_\Gamma A[\sigma] \qquad A \hookrightarrow B \in \Gamma}{M \longrightarrow_{\beta\Gamma} B[\sigma]}\ \textsc{rewrite-reduce} \qquad \frac{AC(F) \in \Gamma \qquad M \sim^*_\Gamma F\ A[\sigma]\ X \qquad A \hookrightarrow B \in \Gamma}{M \longrightarrow_{\beta\Gamma} F\ B[\sigma]\ X}\ \textsc{rewrite-reduce-extended}$$

The new reduction rules REWRITE-REDUCE and REWRITE-REDUCE-EXTENDED are used to make rewrite rules definitional. They require the left hand side (LHS) of a rewrite rule to match the target term or a subset of the target modulo equivalence. This means that rewrite rules on

AC terms will require AC-matching to see if the rewrite rule applies. This check is described in section 3.3.3. The REWRITE-REDUCE-EXTENDED corresponds neatly to rewrite rule extensions as describe by Peterson and Stickel (1981). The difference is that instead of extending the rewrite rules, we extend the reduction relation.

3.3.2 Pattern Fragment

For this thesis, we are not interested in a fully general type-checking procedures for rewrite rules modulo AC. Instead we will focus on a fragment of patterns that is easy to match. The goal is to reuse rewrite rule matching and unification procedures of patterns without AC functions. To achieve this we introduce four restrictions on the LHS pattern of rewrite rules.

1. To prevent matching on potential β-redexes, every application in a pattern has the shape $f\bar{x}$, where $\bar{x}$ is a list of arguments and f is not a free variable. Furthermore, the pattern root must be such an application to make sure that it does not match inside potential β-redexes.

2. AC sub-terms in our patterns must have at most two elements.

   We know which terms are AC, because all applications in a pattern have non free head symbol and for each head symbol we can check if they are AC$_n$ for any n.

   To check if AC sub-terms have at most two elements, we thus need to check that nested AC function applications either do not have the same head-symbol, or are applications of non-unifiable parameters.

3. AC sub-terms that are not at the root of the pattern must not require commutativity or associativity to apply.

   In order to guarantee that associativity is not required for sub-terms, it is enough to restrict AC sub-terms to have at most two elements as before. To make sure that commutativity is not required, it is enough to check that the two elements are identical except for pairs of independent free variables between the arguments.

4. If two sets of free variables are interchangeable in the LHS, then they must also be interchangeable in the right hand side (RHS).

   Two sets of free variables are 'interchangeable' if they can be swapped without changing the LHS modulo AC. Because commutativity must not be required to match AC sub-terms in a pattern, it will be possible to swap the independent free variables between the two elements of these sub-terms. When this happens we must check that swapping the same sets of free variables does not change the RHS modulo AC.

These restrictions make it possible to do pattern matching with $O(n^2)$ complexity, where n is the size of the terms. This is described in section 3.3.3.

3.3.3 Rewriting Rule Matching Modulo AC

To apply rewrite rules, it is first necessary to find where the rules can be applied, this is called rewrite rule matching. Defining AC functions in the context means that rewrite rule matching becomes more complex. This is visible from the REWRITE-REDUCE and REWRITE-REDUCE-EXTENDED inference rules, in which $\sim_\Gamma^*$ is used to compare the pattern term to the target term. Since $\sim_\Gamma^*$ compares terms modulo AC, we also have to do this during pattern matching.

In order for our reduction to be exhaustive, we need to make sure that we apply all possible rewrite rules. We have restricted AC rewrite rules to have at most two elements at the root. This makes matching as simple as finding all pairs of elements in any target AC term. For each pair we use regular pattern matching with normal and swapped elements.

$$\begin{tikzcd}
& B' \arrow[r, "\longrightarrow^*_{\beta\Gamma}"'] & C' \arrow[dd, leftrightarrow, "\sim^*_{\Gamma}"] \\
A \arrow[ur, "\longrightarrow_{\beta\Gamma}"'] \arrow[dr, "\longrightarrow_{\beta\Gamma}"] & & \\
& B \arrow[r, "\longrightarrow^*_{\beta\Gamma}"] & C
\end{tikzcd}$$

Figure 3.2: Local confluence.

$$\begin{tikzcd}
& B' \arrow[r, "\longrightarrow^*_{\beta\Gamma}"'] & C' \arrow[dd, leftrightarrow, "\sim^*_{\Gamma}"] \\
A \arrow[ur, "\longrightarrow^*_{\beta\Gamma}"'] \arrow[dr, "\longrightarrow^*_{\beta\Gamma}"] & & \\
& B \arrow[r, "\longrightarrow^*_{\beta\Gamma}"] & C
\end{tikzcd}$$

Figure 3.3: Global confluence.

Lemma 3.3.1. The algorithm is sound.

Proof. If the target term has only two elements, then this corresponds to the REWRITE-REDUCE rule, and if the target has more than two elements, then it corresponds to the REWRITE-REDUCE-EXTENDED rule. □

Lemma 3.3.2. The algorithm applies all possible rewrite rules modulo AC.

Proof. Any AC term at the root of a LHS pattern must have at most two elements. Thus, the only way that such a pattern can match on an AC term with more than two elements is if some of the elements in the pattern are free variables. For example the patterns $M\ X$ _1 and N _1 _2, where M and N are AC functions, could match an AC term with more than two elements. For every such match it is possible to match just two elements by substituting each free variable with a single element from the elements that it matched previously.

We thus have that it is impossible to have a term that is not completely reduced and cannot be reduced further by matching two elements.

□

3.3.4 Confluence

There are two related confluence properties, local confluence and (global) confluence. Local confluence requires that given two initial reductions $A \longrightarrow_{\beta\Gamma} B$ and $A \longrightarrow_{\beta\Gamma} B'$, there must be some sequence of reductions from each term $B \longrightarrow^*_{\beta\Gamma} C$ and $B' \longrightarrow^*_{\beta\Gamma} C'$ that result in the same term modulo AC: $C \sim^*_{\Gamma} C'$. This is shown in figure 3.2. The stronger property of global confluence requires the same property after any number of initial reductions as can be seen in 3.3.

Local confluence is enough to guarantee that there is at most one normal form for each term, but it does not guarantee subject reduction. Furthermore, neither property is enough to guarantee completeness of our conversion checking procedure. To guarantee completeness we need the rewrite system to be terminating. Termination checking is difficult and that is why we might want to check global confluence so that we can at least guarantee subject reduction. For the purpose of this thesis however, we make termination checking the responsibility of the user. This means that local and global confluence are identical and both guarantee completeness and subject reduction.

3.3.5 Local Confluence Checking

This section describes how we can check local confluence as described in 3.3.4. To guarantee local confluence we have to consider all pairs of reductions. Rewrite rules can not match at potential β-redexes, so we know that at least rewrite rules are confluent with β-reduction. However, we do not know if rewrite rules are confluent with each other. In order to guarantee this, we first need to find all critical pairs. As stated by the REWRITE-REDUCE and REWRITE-REDUCE-EXTENDED rules, we are allowed to use the equivalence relation to apply rewrite rules. For this reason the definition of critical pairs depends on overlap and the definition of overlap uses the equivalence relation.

Definition 3.3.6. We define $Pos(X)$ to be the set of all positions of sub-terms inside the term X. It is possible to replace a sub-term at position p in X with another term Y using the notation $X_{p|Y}$.

Definition 3.3.7. Let there be two patterns X and Y and a position $p \in Pos(X)$. We have that X *overlaps* Y at position p if there exists substitutions σ_1 and σ_2 such that $X[\sigma_1] \sim_\Gamma^* X_{p|Y}[\sigma_2]$. If we also have two rewrite rules $X \hookrightarrow X'$ and $Y \hookrightarrow Y'$, then we have a critical pair $(X'[\sigma_1], X_{p|Y'}[\sigma_2])$.

Definitionally AC function allow for a new kind of overlap between patterns that we call partial overlap:

Definition 3.3.8. Let there be two patterns X and Y and two positions $p \in Pos(X)$ and $q \in Pos(Y)$. We have *partial overlap* between these two patterns if there exists substitutions σ_1 and σ_2 such that $Y_{q|X}[\sigma_1] \sim_\Gamma^* X_{p|Y}[\sigma_2]$. If we also have two rewrite rules $X \hookrightarrow X'$ and $Y \hookrightarrow Y'$, then we have a critical pair $(Y_{q|X'}[\sigma_1], X_{p|Y'}[\sigma_2])$.

Definition 3.3.9. A *critical pair* is the result of applying two rewrite rules to the unification of their LHSs with some overlap as defined in 3.3.7 and 3.3.8.

Definition 3.3.10. A critical pair (A, B) is *more general* than another critical pair (X, Y), if there exists a substitution σ such that $(A[\sigma], B[\sigma]) = (X, Y)$ or $(A[\sigma], B[\sigma]) = (Y, X)$.

Definition 3.3.11. A *complete set* of critical pairs contains for each critical pair at least one more general critical pair.

For the purpose of this thesis, we assume that there is already a 'normal' unification procedure implemented, which handles AC terms as if they have no extra equivalences. This unification procedure is also guaranteed to return the most general unifier, which means that the associated critical pair is more general than any other critical pair for the same unification problem.

The definition of critical pairs above uses overlap which uses the equivalence relation. This meant that in order to naively find critical pairs we would have to do AC unification. We show that for our pattern fragment and calculus, we can actually find a complete set of critical pairs using the 'normal' unification procedure. First we will find all critical pairs with full overlap and then those with partial overlap.

Full Overlap at Internal Position

We know that rewrite rules in our fragment are not allowed to require AC rules for matching at positions other than the root. We also know that interchangeable free variables in the LHS must be interchangeable in the RHS. These two restrictions together mean that we do not have to consider AC rules when creating critical pairs at positions other than the root.

This means that if we have two rewrite rules $X \hookrightarrow X'$ and $Y \hookrightarrow Y'$ and some position $p \in Pos(X)$. Then we can always create at most one critical pair, no matter if some term at the root of Y is AC. If unification gives us some σ_x and σ_y such that $X[\sigma_x] = X_{p|Y}[\sigma_y]$, then we have critical pair $(X'[\sigma_x], X_{p|Y'}[\sigma_y])$.

Full Overlap at the Root

We assume that we are creating critical pairs for two rewrite rules $X \hookrightarrow X'$ and $Y \hookrightarrow Y'$. We can have up to two critical pairs with full overlap:

- Whether or not our rewrite rules have an AC term at the root, we need to try creating a critical pair by just unifying X with Y. Formally, if we have $X[\sigma_x] = Y[\sigma_y]$ for some σ_x and σ_y, then the critical pair is $(X'[\sigma_x], Y'[\sigma_y])$.
- If a term at the root of our rewrite rules is AC, we need to unify X with every AC pattern M and swap the arguments before unifying with Y. Formally, if we have $M[\sigma_m]\ A\ B = X[\sigma_x]$ and $M[\sigma_m]\ B\ A = Y[\sigma_y]$ for some A, B, σ_m, σ_x and σ_y, then the critical pair is $(X'[\sigma_x], Y'[\sigma_y])$.

Definition 3.3.12. An AC pattern M is an AC$_n$ constant f applied to n independent free variables. This would look like $f\ \_1\ \cdots\ \_n$.

Partial Overlap at the Root

Again we assume that we are creating critical pairs for two rewrite rules $X \hookrightarrow X'$ and $Y \hookrightarrow Y'$. There is potential for partial overlap in case both rewrite rules have an AC term at the root. For every AC pattern M there are four cases, corresponding to the combinations of left and right arguments overlapping between the two patterns. Formally, we need the following two conditions for some $A, B, C, \sigma_m, \sigma_x$ and σ_y:

- $M[\sigma_m]\ A\ B = X[\sigma_x]$ or $M[\sigma_m]\ B\ A = X[\sigma_x]$
- $M[\sigma_m]\ B\ C = Y[\sigma_y]$ or $M[\sigma_m]\ C\ B = Y[\sigma_y]$

If we have both of these conditions, then the rewrite rules can overlap at B and we have the critical pair $(M[\sigma_m]\ C\ X'[\sigma_x], M[\sigma_m]\ A\ Y'[\sigma_y])$.

Confluence

Now that we know how to compute a complete set of critical pairs, all that remains is to check local confluence for each critical pair. Local confluence only require that the two terms convert to one another. This is checked using the conversion checker modulo AC as described in section 3.2.1.

Lemma 3.3.3. If every critical pair in a complete set is confluent, then the reduction relation is confluent.

Proof. There can not be any critical pairs between rewrite rules and β-reduction, because rewrite rule patterns do not match at potential β-redexes. Rewrite rules that apply to the same term without any overlap are trivially confluent. This is because applying one rewrite rule can not influence the applicability of the other rewrite rule and the results are equal because of congruence. Finally, it is only necessary to check the most general critical pairs, because the confluence of other critical pairs follows from the more general ones. □

Chapter 4

Implementation

This chapter explains how the ideas from the core calculus presented in Chapter 3 have been applied to Agda. The Agda compiler is a big project, so we won't describe how the parts work that are irrelevant to this thesis. This is a high level overview of the steps used for type-checking:

- The source code is parsed into an abstract syntax tree (AST).
- The AST is desugared into a core AST.
- The core AST is type-checked.
  - Bidirectional type-checking is used to gather constraints.
  - Constraints are solved using the conversion checker.

First we give some background knowledge on type-checking in Agda and the scope of implementation in section 4.1 and 4.2. Then we continue with descriptions of the different procedures used for type-checking that are affected by the addition of definitional AC functions in section 4.3, 4.4 and 4.5.

4.1 Agda Type Checking Overview

Agda type-checking extends bidirectional type-checking with meta-variables. Meta-variables are used for implicit arguments as shown in section 2.1. When calling a function, all implicit arguments are substituted with meta-variables.

Bidirectional type-checking is only the first step in type-checking with meta-variables. It is used to gather conversion constraints that are later solved by the conversion checker. Conversion constraints are pairs of terms that should convert to one another. Terms in these constraints can contain meta-variables, and this can block conversion checking. When this happens, the conversion checker moves on to the next constraint and when more meta-variables are solved it can retry constraints that have previously failed.

The type-checker is allowed to give each meta-variable a value as required, but only if it is guaranteed that all solutions are definitionally equal. So for example if $a + 0$ and a are the only solutions, and we have $a + 0 \longrightarrow^*_{\beta\Gamma} a$, then they are both valid substitutions.

We do not solve meta-variables that have multiple solutions, because the number of solutions can reduce after solving other meta-variables. In this case we might need to backtrack and essentially try every solution. This is infeasible for meta-variables with many solutions.

Remark. When we need to write down meta-variable names in this thesis, we prefix them with question marks (e.g., $?1, ?2$).

Core AST Representation

Agda uses spinal β-normal form to represent terms in its core AST. This means that not only are there no β-redexes, but multiple applications have been combined into a single function symbol with a list of arguments. As an example, the application `f a b c` would not be represented with three `APP(func, arg)` nodes, but instead a single `APP(func, arg_list)`. The `func` in such nodes is called the head symbol and can only be a variable or constant. This is because if `func` is an application then the term is not spinal, and if `func` is an abstraction then the term is not β-normal. β-normal form makes it possible to type-check with bidirectional typing and spinal form makes the representation more efficient and easier to reason about.

4.2 Scope of Implementation

Even with the concise theory that we want to implement, there was not enough time to implement everything. The two missing features are confluence checking and parameterised AC constants. Both of these features are not necessary for the examples. The next chapters will describe how the theory was implemented.

4.3 Normalization

Normalizing terms is important for the type-checker, it is used a lot in the conversion checker. We already know how normalization works with definitionally AC function as described in chapter 3. Agda has a second normal form however, it is called weak head normal form (WHNF) and is used because it is much cheaper to compute.

Weak Head Normal Form

An application term is in WHNF if the head is completely normalized. This means that no rewrite rule is allowed to match on the root of the term. This has the convenient result that the arguments of the term head can no longer interact. Interaction between arguments here means that it is possible to have a rewrite rules match on two arguments at once.

When the term head is AC, this definition does not really change and the representation of terms does not change either. However, now we want all elements of the AC term to no longer interact. We thus need to check that no rewrite rule can ever apply to any pair of elements as described in section 3.3.3.

Lemma 4.3.1. The head symbol of a term in WHNF can be assumed to be injective with respect to definitional equality.

Proof. By contradiction: assume we have some definitional equality $f\ \bar{a} = f\ \bar{b}$, where both terms are in WHNF and the argument lists are not definitionally equal $\bar{a} \neq \bar{b}$. For the two terms to be definitionally equal, we must have some reduction from either term. However, both terms are in WHNF, so there can be no such reduction. Thus we have a contradiction and the only possible conclusion is that $f\ \bar{a} = f\ \bar{b} \implies \bar{a} = \bar{b}$. □

Normalization Procedures Modulo AC

Agda has two important procedures for evaluation, `Reduce` and `Normalize`. `Normalize` normalizes terms to their normal form and `Reduce` only normalizes to WHNF. Both procedures follow similar steps, the only difference is that `Reduce` unflattens after AC matching, because at this point the head symbol of the term is known and the elements can no longer interact. The steps of normalizing an AC term can be seen in figure 4.1 and every step is explained below:

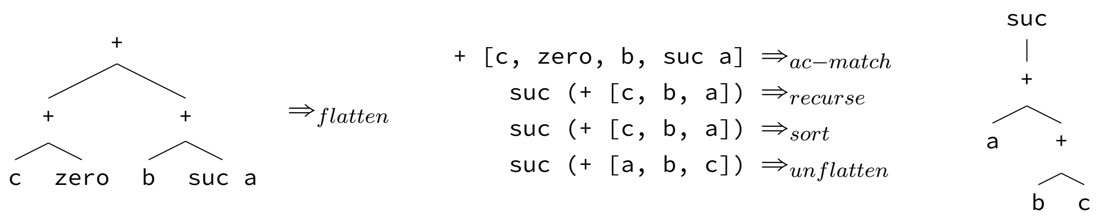

+
+
+
c
zero
b
suc a
⇒flatten
+ [c, zero, b, suc a] ⇒ac−match
suc (+ [c, b, a]) ⇒recurse
suc (+ [c, b, a]) ⇒sort
suc (+ [a, b, c]) ⇒unflatten
suc
+
a
+
b
c

Figure 4.1: Normalising an AC-term.

- *Flattening* an AC term gathers all the elements into a list. This requires comparing the head symbol of the root with the head symbol of each argument recursively. Checking the head symbol of an argument requires reducing first. Since Agda has meta-variables, we might not be able to flatten a term until some meta-variables have been resolved.

- *AC matching* every pair of elements is necessary to make sure that the elements do no longer interact. This is where the `Reduce` procedure would stop and unflatten the elements back into a tree.

- *Recursively* normalizing the elements needs to be done before sorting in order to make sure that the syntactic order is consistent.

- *Sorting* the elements in an AC term is done syntactically. It is important to not use any irrelevant details on the terms for sorting, because this might make equal elements non-adjacent.

- *Unflattening* puts the elements back in a tree structure. This tree is made right-associative and uses the order from the sorted list. This means that the tree is canonical just like the sorted list.

It is not possible to sort argument in the `Reduce` procedure, because the sorting might not be correct. This means that the element order as returned by `Reduce` can not be trusted. We have to call `Normalize` explicitly in all places where the term head is AC and the order of the elements matters.

4.4 Conversion Checking

Conversion checking is the most important procedure that needs to be modified when adding support for definitional AC functions. So in this section we give an overview of modifications that need to be made.

4.4.1 General Conversion Checking Procedure

We first describe how conversion checking with meta-variables works without definitional AC functions:

- The simplest case in the conversion checker is when one of the terms it tries to convert is a meta-variable. In this case the meta-variable is immediately instantiated with the other term.

- If neither term is a meta-variable then we distinguish three cases based on how many of the terms can be reduced to WHNF:

  - When both terms reduce to WHNF, the only way type checking can succeed is if between both sides the heads are equal and the arguments convert. This is because WHNF guarantees that the head of the term can not change and can be assumed to be injective with respect do definitional equality.
  - When neither term reduces to WHNF, then the conversion checker checks if the head and arguments can be converted between both terms without solving any meta-variables. If it succeeds then we know that both sides are indeed equal and we have not lost any solutions. Allowing the conversion checker to solve meta-variables here could lose solution because we can not assume the term heads to be injective.
  - Agda supports one more case, which is when one term can be reduced to WHNF and the other term can not. A term that can not be reduced to WHNF is called 'blocked'. In this case injectivity analysis can sometimes still refine the shape of meta-variables in the blocked term or solve them completely. How this worked will be described in section 4.5.

4.4.2 Conversion Check Modulo AC

Now that we know how conversion checking with meta-variables works in Agda, we can describe the changes that need to be made to support AC functions.

All the same rules from regular conversion checking apply. The only difference is that in the case where both terms reduce to WHNF with equal head symbol that is AC, we can no longer just conversion check the arguments. What we do instead is flatten both AC terms into element lists. Because both sides are in WHNF, we know that the elements can not interact. Now if both sides have at least two elements each, we proceed with a new algorithm:

First we normalize both element lists and remove equal elements between both sides. Note that both lists always start with at least two elements before removing equal elements. Afterwards we will have one of the following cases:

- Both element lists are empty, in this case the two terms were equal and conversion succeeds.
- One of the element lists is empty, but the other list is not. In this case type-checking has failed. This is because there can not be a definitional unit element. The existence of such an element would prevent the larger term from being in WHNF. Thus there is no solution for the non empty element list.
- One of the element lists has one element and the other list is not empty. In this case we check conversion of the single element with the AC term constructed from the other element list. This does not cause infinite recursion in the conversion checker, because the single element is a subset of the original element list.
- Both sides have more than two elements. In this case we do not know how to match the elements to each other without more in depth analysis. We are stuck on this constraint and proceed with the next constraint.

4.5 Injectivity Analysis

It turns out that when you no longer have to prove AC equalities, the most annoying thing is that you need to specify the values of many arguments that could be inferred. This is where injectivity analysis is very important, and that is why we dedicated quite some time to figuring out how injectivity analysis works and how to make this better. The current injectivity analysis that is implemented in Agda cannot handle rewrite rules. So what we do here is describe how it is made to work with rewrite rules. In appendix A, we will describe more enhancements that make the injectivity analysis able to infer more meta-variables.

4.5.1 Injectivity Analysis with Rewrite Rules

We will first describe injectivity analysis with rewrite rules, but without definitional AC functions. Note that injectivity analysis is part of conversion checking and if this procedure gets stuck at any point the type-checker will continue with the next constraint as described in 4.1.

Remember that injectivity analysis is applied in the conversion checker when one side is in WHNF and the other side is blocked on some meta-variable. We discern two cases here: either the term head symbols are equal, or they are not. These cases have different solutions as described below:

Injectivity Analysis without Equal Heads

Because the heads of both terms are not equal, we know that some reduction must happen on the blocked term in order to make conversion checking succeed. To decide which reductions are possible, the injectivity analysis looks at the RHSs of the computation rules of the blocked term. Some RHSs might have an unknown head, in that case we assume every head is possible. If only one rule has the correct head as possibility in the RHS, then that is the computation rule that must execute. We can refine the meta-variables in the blocked term to make the blocked term match the computation rule LHS. If more than one rule has the correct head as a possibility in the RHS, then we do not know which rule needs to execute and we are stuck. Finally, if no rule can possibly rewrite to the correct head, then we know type-checking has failed.

Injectivity Analysis with Equal Heads

When the head of both terms is equal, but one side is blocked. Then we can not assume that a reduction is necessary. We can thus also not refine the blocked side to something that reduces. In this case, we check if it is possible for the blocked side to reduce to a term with the same head. If this is not possible, then the only way both sides are going to be equal is if all the arguments are equal. In this case we can proceed by conversion checking the arguments. If instead it is possible for the blocked term to reduce to a term with the same head, then we are stuck.

4.5.2 Injectivity Analysis modulo AC

Injectivity analysis with definitional AC functions is much more difficult:

Without Equal Heads

The only way for an AC term to produce a different head symbol is if the list of elements reduces to just two elements. This is because rewrite rules are only applied to two elements at once as described in 3.3.3.

If the AC term has exactly two elements, then we can assume that one of the rewrite rules needs to be applied in normal or swapped order. If there are more than two elements, then we do not know if the current split of elements over the arguments at the root of the AC term reduces to the two elements that we need. Thus we can not easily apply injectivity analysis in this case and we are stuck.

With Equal Heads

With equal heads there is the possibility that no reduction is necessary at all. We would like to do something similar to conversion checking with two terms in WHNF. It is however not easy to guarantee that this is allowed. How to do this is left as future work. For now we are just stuck when checking injectivity of an AC term with equal head symbols.

Chapter 5

Usage Examples

This chapter is about reviewing the theory and Agda implementation to see if it is possible to automatically write proofs involving commutativity and associativity. We prove some properties of sorting functions in section 5.1, define a `Bag` data-structure for natural numbers in section 5.2 and finally implement a simple effect system with effect rows in section 5.3.

5.1 Sorting Algorithms

There are two approaches to formally verifying functions in a proof assistant: the extrinsic approach and the intrinsic approach. Here we demonstrate both approaches for a sorting function on lists. The extrinsic approach is to have a generic signature and then have separate proofs like this:

```
sort : List Nat → List Nat
sort = ?

preserves-items : (list : List Nat) → items list ≡ items (sort list)
preserves-items = ?

sorts-items : (list : List Nat) → is-sorted (sort list) ≡ true
sorts-items = ?
```

The proofs and the function can be bundled together to make the function interchangeable with other sorting function. The tuple that bundles these things together would look like this: `(sort, preserves-items, sorts-items)`. Note that it is also possible to ignore the proofs and treat the sorting function as a more abstract function on lists.

The intrinsic approach to formally verify a sorting function is with intrinsic data-types. Lets leave `sorts-items` for now and only verify `preserves-items`: we want to model that the elements in the input list are the same as the elements in the output list. The data-structure that contains just information about included items is the multi-set. We thus define a multi-set for natural numbers and call it `Bag`. Now we can add an implicit argument to our sorting function that specifies the included elements so that we can refine the input and output list to have these elements. This refinement requires a list data-structure that is indexed by a `Bag`, which we will call `UnSorted`. Taking all of this together results in the following type for sorting functions:

```
postulate
  Bag : Set
  ∅ : Bag
  bag : Nat → Bag
  -- disjoin union: returns a bag containing all items from both bags
  _⊔_ : Bag → Bag → Bag

  ∅⊔xs : (xs : Bag) → ∅ ⊔ xs ≡ xs
  xs⊔∅ : (xs : Bag) → xs ⊔ ∅ ≡ xs

  ⊔-comm : (xs ys : Bag) → xs ⊔ ys ≡ ys ⊔ xs
  ⊔-assoc : (xs ys zs : Bag) → (xs ⊔ ys) ⊔ zs ≡ xs ⊔ (ys ⊔ zs)

data UnSorted : Bag → Set where
  [] : UnSorted ∅
  _::_ : (x : Nat) {xb : Bag} → UnSorted xb
       → UnSorted (bag x ⊔ xb)

sort' : {xs : Bag} → UnSorted xs → UnSorted xs
sort' = ?
```

All implementations of this sorting function must preserve the items exactly. This is checked by the type-system and does not require separate proof. The sorting function is also immediately interchangeable with other functions that preserve items. However, if we want to use this function in the place of an abstract function on lists, then we must write adapters to and from our indexed `UnSorted` to normal `List`.

For the implementation of the `sort` function to be easy, it is essential that `Bag` has definitionally AC disjoint union (⊔). To see why, let us look at an insert function that one might use to implement `sort`. Without definitionally AC disjoint union, it would look something like this:

```
swap-left : ∀ x y ys → y ⊔ (x ⊔ ys) ≡ x ⊔ (y ⊔ ys)
swap-left = ?

insert : (x : Nat) → {xs : Bag Nat} → UnSorted xs → UnSorted (bag x ⊔ xs)
insert x [] = x :: []
insert x (y :: ys) = case x < y of \where
  true → x :: y :: ys
  false → cast swap-left (y :: insert x ys)
```

Note that it requires proving and invoking a new theorem `swap-left`. This can be left to the type-checker with the AC pragma:

```
{-# COMMASSOC ⊔-comm ⊔-assoc #-}

insert' : (x : Nat) → {xs : Bag Nat} → UnSorted xs → UnSorted (bag x ⊔ xs)
insert' x [] = x :: []
insert' x (y :: ys) = case x < y of \where
  true → x :: y :: ys
  false → y :: insert' x ys
```

If we want to intrinsically check `sorts-items`, then we need a new `Sorted` list type that refines the elements in the `Bag`. How to refine elements in a bag is described in section 5.2. Appendix C.1 has a fully verified implementation of two sorting algorithms: Insertion sort and merge sort. These algorithms have been implemented without any proof writing that is specific to each sorting algorithm. The only things that were proven were general properties of `Bag` and comparisons. This shows that it is a viable development strategy of which I hope to see more in the future.

5.2 Bags

When designing a `Bag` data-structure, the most important properties for usability are the supported operations, rewrite rules and injectivity analysis. That is why this chapter will focus on these aspects.

5.2.1 Mapping Bags

An important design consideration is support for refining the elements of the bag as a whole. For example, in the case of sorting algorithms, it can be very useful to refine elements in a bag to be lower bounded by some number. This can be achieved by refining the numbers to be additions of the lower bound and some new value. For example `y ≡ x + d` could be used to refine a single number `y`. To refine all elements in a `Bag`, we would need a `map` function, so that we can write `xs ≡ map (\d → x + d) ds`. If we define a map function for `Bag` however, we must make sure that `map` is reduced away when possible. This requires the following propositional equalities as rewrite rules:

```
postulate
  map : (Nat → Nat) → Bag → Bag
  rew1 : ∀ f xs ys → map f (xs ⊔ ys) ≡ map f xs ⊔ map f ys
  rew2 : ∀ f x → map f (bag x) ≡ bag (f x)
  rew3 : ∀ f → map f ∅ ≡ ∅
```

The problem with these rewrite rules is that the conversion checker can not find the solution for a constraint like `map f xs ⊔ map f ys = map f ?1`. It can use injectivity analysis to refine the meta-variable with `?1 ≡ ?2 ⊔ ?3`, but this will then reduce with rule `rew1` and we end up with the constraint `map f xs ⊔ map f ys = map f ?2 ⊔ map f ?3`. This constraint has multiple solutions and can thus not be solved without the risk of losing the right solution.

What happened is that splitting the meta-variable lost the information that the two meta-variables are always used together and thus it doesn't matter which side we assign to which value. Keeping track of which variables are interchangeable would allow the conversion checker to assign a value to each in any order it wants.

It seems there is a balance, more conversion rules means that injectivity analysis becomes harder. This is why definitional AC breaks injectivity analysis here. The analysis would have worked if ⊔ was not definitionally AC.

A similar problem to the previous one occurs when we add the rewrite rule `map f (map g xs) ≡ map (\x → f (g x)) xs`. This rewrite rule can rewrite `map f ?1` to some `map f ?2`, preventing injectivity analysis on equations like `map f x = map f ?1`. The only way we can know that `map f` is injective is if `f` itself is injective. This is yet another check that could be implemented in injectivity analysis.

5.2.2 Nat Bag Without Postulates

Even though we only care about the rewrite rules and injectivity analysis of a `Bag` for usability, it is useful to have an implementation that is not postulated. This would prove that the data-structure is not inconsistent, which is why we have implemented a bag of natural numbers without postulates.

A bag of natural numbers is useful for sorting algorithms, but at a minimum it needs to have a `raise` function or constructor. `raise` is equivalent to the `map`, but it can only be used to add a value to all elements to create a lower bound. For example, one might want to write `raise x ?1` for a bag that is lower bounded by `x`. It would be best if `raise` was a constructor, because that would mean that all bags reduce to have a lower bound.

The other design consideration for bags is that there needs to be an empty bag. This means that `raise` can not be a `data` constructor, because it is not injective for bags that can be empty. This gives us the following design:

```
data Bag : Set where
  -- the empty bag
  ∅ : Bag
  -- add zero to the bag and raise with the specified Nat
  _↑_ : Nat → Bag → Bag

raise : Nat → Bag → Bag
raise x ∅ = ∅
raise x (y ↑ ys) = (x + y) ↑ ys
```

It can model empty bags and `raise` is defined as a function. Sadly this means that the reduced form of bags does not have a nice lower bound. To make it work satisfactorily we need the following rewrite rules:

```
variable
  x y d : Nat
  xs ys : Bag
postulate
  rew1 : x ↑ xs ⊔ (x + d) ↑ ys ≡ x ↑ d ↑ (xs ⊔ ys)
  rew2 : raise x xs ⊔ (x + d) ↑ ys ≡ raise x (xs ⊔ d ↑ ys)
  rew3 : raise (y + d) xs ⊔ y ↑ ys ≡ y ↑ (raise d xs ⊔ ys)
  rew4 : raise x xs ⊔ raise (x + d) ys ≡ raise x (xs ⊔ raise d ys)
```

Because `raise` is not a constructor, every combination of `raise` and ↑ needs to be handled separately. If addition is marked as AC, then all of these are multi-level AC rewrite rules that are not part of our rewrite rule fragment. If we want to work around that then we can define custom addition which is not AC for use with `raise` and add more rewrite rules for when `d` is zero.

Implementation Using Bag1

An alternative implementation is to first define a `Bag1` data-structure that is not empty. This can then be used to define `Bag` as either `Bag1` or the empty bag:

```
data Bag1 : Set where
  -- bag with just the zero element in it
  [zero] : Bag1
  -- add the zero element to the bag
  zero : Bag1 → Bag1
  -- increment all elements in the bag by one
  suc : Bag1 → Bag1

data Bag : Set where
  -- the empty bag
  ∅ : Bag
  -- the non-empty bag
  just : Bag1 → Bag
```

The constructor `suc` is like `raise 1` and `zero` adds another zero element to the bag. `raise` is still not a constructor, but the advantage of this implementation is that Agda can automatically check termination of the disjoint union function. The full source code for this data-structure is provided in appendix C.2.

5.3 Effect Systems

The final use case that we explored is effect systems. In pure type systems it is common to model side effects through monads and `do` notation. Multiple effects can be used at the same time with a stack of monad transformers, but when many effects are used at the same time this can become unwieldy. Effect rows are an alternative to these monad stacks that make it easier to work with many effects by making them unordered.

Previous work by Bach Poulsen and Rest (2023) has used overlapping instance arguments to implement effect rows[1], but we will show here that this is also possible with the AC pragma. The implemented effect system only aims to demonstrate the composition of effects and that they are unordered. The full source code for this is provided in the appendix C.3.

```
-- The universe of two writer effects
data Eff : Set where
  Bools : Eff
  Nats : Eff

-- embedding of universe into types
get-eff : Eff → Set
get-eff Bools = List Bool
get-eff Nats = List Nat

-- Encoding of required Eff in the environment
Cond : Set
Cond = List Eff → Bool

-- data representing effectfull computation
data WithEff : ∀ {L : List Eff} (v : Bool) (O : Set) → Set₁ where
  pure : ∀ {v O} → O → WithEff {[]} v O
  emit : ∀ {L E v} → get-eff E → WithEff {E :: L} v ⊤
  inc : ∀ {L E v O} → WithEff {L} v O → WithEff {E :: L} v O
  _>>=_ : ∀ {L v X O} → WithEff {L} v X → (X → WithEff {L} v O) → WithEff {L} v O
  fail : ∀ {L O} → WithEff {L} false O
```

How it works is that `WithEff` is indexed by an environment of available effects and the result of some `Cond` on that environment. This prevents problems from green slime by moving all problems into the `fail` constructor. After everything is worked out we can write code like this:

```
test : WithEff' (eff Nats && eff Bools) ⊤
test = do
  smart-emit {Nats} (10 :: [])
  smart-emit {Bools} (true :: [])
  smart-emit {Nats} (42 :: [])

main : List Nat
main = eval do
  (nats , bools) ← handle {Bools} do
    (tt , nats) ← handle {Nats} test
    smart-pure nats
  smart-pure nats
```

[1] https://github.com/heft-lang/POPL2023/blob/master/src/Free.agda

```
main1 : WithEff' (eff Bools) (List Nat)
main1 = do
  (tt , nats) ← handle {Nats} test
  smart-pure nats

test2 : WithEff' (eff Nats && eff Bools) ⊤
test2 = do
  _ ← smart-raise main1
  smart-pure tt
```

Normally whenever you would used `smart-emit`, the AC checker would fail to find what to use for the remainder of the effects. Our workaround is to use the fact that the output type of the function can almost always be inferred from its call-site. The requirements of the output are stored in `v` and we require an instance argument to prove that `eff E xs` is a subset.

```
smart-emit : ∀ {E v xs} → {{v &&' eff E xs ≡ v}} → get-eff E → WithEff {xs} v ⊤
smart-emit {E} {v} {xs} {{p}} e = cast (λ x → WithEff x ⊤) p (emit-idx {v} (idx E xs) e)
```

Chapter 6

Related work

This section reviews some of the related work. We are interested in alternatives that automate reasoning about AC functions. These works are organized by their method of implementation.

6.1 Implemented in the Type System

The paper about CoqMT (and CoqMTU) proves that it is possible to extend an impredicative type-system with universe levels and inductive data-types, such as Coq, with first order theories. The paper does not explain how to extend an implementation with such a theory, but the repository[1] does have the theory of natural numbers implemented for Coq. Thus it has a good theoretical result, but does not describe how to let users add their own theories. Rewrite rules and AC definitions from this thesis aim to define and implement a system for users to use the theory that they want.

Dedukti also has some definitional AC functions. While we were not able to find any documentation, the source code is available[2].

6.2 Non-Dependent Type Systems

ELAN (Borovanský et al. 2002) is a logic-based rewriting system with support for AC function symbols and conditional rewriting. Other related systems are Maude 2.0 (Clavel et al. 2002) and CafeObj (Diaconescu and Futatsugi 2002), both of which have support for AC function symbols. While it is possible to prove properties in these systems, they are not dependently typed and can thus not be used as generally as the type system described in this thesis.

6.3 Implemented with Macros

Another project that is related to ours is Frex (Allais and Brady 2022), it is a library and the approach can thus be used for any dependent language. Their approach is to let users define the simplifiers for any theory and it will give you procedures to automatically write proofs about those theories. The simplifiers can be build modularly and are verified by the type-checker to be sound and complete. This is an alternative approach to rewrite rules that does not require confluence checking. The approach used by Frex is able to handle associativity and commutativity, because it represents terms with a data structure that allows ordering terms. The biggest downside is that it requires explicit invocation of the macro and it is also

[1]`https://github.com/strub/coqmt`

[2]`https://github.com/Deducteam/Dedukti/blob/abbe11318189c32ae236671ccd8dcf3c89525065/kernel/ac.ml`

not clear how one would make rewrite rules match on terms that are only AC inside the macro.

	Agda modulo AC	Agda Macros (Frex)
AC conversion	automatic	invoked
AC rewrite rule matching	yes	no
User defined theories	no	yes
Meta variable inference	partial	?

Adding AC to the type-checker has the primary benefit that it does not require invocation of macros like Frex and it allows interaction with rewrite rules. A macro system has the benefit that it allows users to define their own theories. It is unclear how well macros can infer meta-variables, but it is likely to be worse than what is possible in the type-checker. In conclusion, both approaches are useful and it would be interesting to see if their advantages can be combined.

Chapter 7

Discussion

From chapter 5 we know how the new AC pragma in Agda can be used and this chapter will discuss the results. There is also a list of future research directions that we think could be interesting.

Theory

Adding AC function equalities to $\lambda\Pi$ was not that difficult and fits really well as can be seen in chapter 3. By splitting the conversion check into a reduction and equivalence relation, everything stays neatly organized. The starting formalization of $\lambda\Pi$ uses the context for constants and this is easily extended with AC neutral terms and rewrite rules. Adding rewrite rules is useful because it makes $\lambda\Pi$ similar in expressiveness to other dependently typed languages. We were able to describe a fragment of rewrite rules that are easy to apply and to check confluence of. This greatly helped with the next step of implementing AC functions in Agda.

Implementation

Adding AC conversion checking to Agda is doable with the restrictions that we described in section 3.3.2. The restrictions allow a simple wrapper for existing rewrite rule matching with quadratic run-time in the size of AC terms. To support rewrite rules outside of our fragment it would be necessary to modify the `NonLinMatch` procedures in Agda which is quite complicated. It would also require more advanced algorithms to keep up the matching speed for complicated rewrite rules. Although the scope of our implementation was slightly limited as described in section 4.2, the implementation is complete enough to type-check our examples. The final implementation is thus a minimum viable product that should only be used to check if this feature is something that is desired.

The biggest hurdle in the implementation was integration with meta-variables. Agda has a system for meta-variables that is undecidable. This is because it requires higher order unification which has been proven undecidable (Spies and Forster 2020). We noticed that it is possible to put a lot of time in improvements to the inference of meta-variables. Some examples from this thesis are improvements to the injectivity analysis and elimination of common AC sub-terms in the conversion checker. This was not part of the core calculus and is not essential, since one can always provide explicit arguments instead of relying on meta-variables. Thus for future research I think it is important to limit the scope of meta-variable inference.

Usage Examples

As shown in chapter 5, the theory of AC functions is very useful for some use-cases. Where previously one would need to use overlapping instances or macros, AC reasoning can now be done automatically by the type-checker. Even though examples have been shown regarding sorting algorithms and effect systems, there is still a lot of research needed to find all the possible applications and limitations. We imagine that with support for definitional AC functions in the type-checker it is viable to implement a full effect system as a library . Another use-case that was not explored in this thesis is a simpler implementation of 'data types à la carte' (Swierstra 2008).

7.1 Future work

This is a list of research directions and improvement to the current implementation that are left as future work:

- The Agda implementation completely normalizes AC terms in order to compare them. Instead, it should be possible to normalise them just enough for comparison. It could even make sense to refactor the conversion checker to return an ordering between its input terms.
- The AC pragma in Agda only works with `--no-fast-reduce` flag. This flag turns the Agda Abstract Machine (AAM) off. Future work could integrate AC conversion checking into the AAM.
- With the current implementation it is only possible to mark a function as being both commutative and associative at the same time. It would be preferable if these properties had separate pragmas that could be combined. Solving this in a modular way would allow adding more properties like unit element or idempotence. The importance of which in the presence of rewrite rules is also unclear.
- Similarly, there is currently no support for commutativity of `cons`-like operators. These are operators that add an element of type `A`, to a structure of type `B`, and return a structure of type `B`. While operators like this are not commutative or associative, they can behave like multi-sets in the same way that AC functions do. Maybe this is not that important, because it is often possible to turn elements into singleton data structures that can then be combined using a normal commutative operator.
- The current implementation only supports definitional AC symbols. However, the theory presented in this thesis allows parameterised symbols to be definitionally AC. This would be useful for a polymorphic `_⊔_ : (A : Set) -> Bag A -> Bag A -> Bag A` that is defined to be AC for all `A`. The current implementation also doesn't support AC terms that return a function. This would be something like `_&&_ : (Bag -> Bool) -> (Bag -> Bool) -> (Bag -> Bool)`. It should not be too difficult to extend the current implementation to support these.
- It might be possible to use the new AC support to replace the universe level machinery[1]. This would simplify the Agda implementation. In theory all that is necessary is to mark the maximum operator on universes as AC and remove the special case code from the compiler.
- It could be useful to explore an alternative direction where you have A builtin data-structures for multi-sets. This builtin multi-set can then be used as function input to indicate that the function is associative and commutative. This would remove the need for the `COMMASSOC` pragma, but it is unclear how you would pattern match on this builtin multi-set.

[1] `https://agda.readthedocs.io/en/v2.6.3/language/universe-levels.html`

Bibliography

Allais, Guillaume and Edwin C. Brady (2022). “Frex: dependently-typed algebraic simplification”. In: URL: https://www.semanticscholar.org/paper/Frex%3A-dependently-typed-algebraic-simplification-Allais-Brady/169245998d755fc319f53348e2c9c3f11234b8bb (visited on 02/02/2023).

Altenkirch, Thorsten, Conor McBride, and James McKinna (2005). “Why Dependent Types Matter”. en. In.

Bach Poulsen, Casper and Cas van der Rest (Jan. 2023). “Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects”. In: *Proceedings of the ACM on Programming Languages* 7.POPL, 62:1801–62:1831. DOI: 10.1145/3571255. URL: https://dl.acm.org/doi/10.1145/3571255 (visited on 07/07/2023).

Blanqui, Frédéric (Oct. 2020). *Type safety of rewrite rules in dependent types*. arXiv:2010.16111 [cs]. DOI: 10.4230/LIPIcs.FSCD.2020.13. URL: http://arxiv.org/abs/2010.16111 (visited on 03/09/2023).

Boespflug, Mathieu, Quentin Carbonneaux, and O. Hermant (2012). “The λΠ-calculus Modulo as a Universal Proof Language”. In: URL: https://www.semanticscholar.org/paper/The-%CE%BB%CE%A0-calculus-Modulo-as-a-Universal-Proof-Boespflug-Carbonneaux/56419ea95a4ebbaa7128b489f7f551b006eb5616 (visited on 04/13/2023).

Borovanský, Peter et al. (Aug. 2002). “ELAN from a rewriting logic point of view”. en. In: *Theoretical Computer Science*. Rewriting Logic and its Applications 285.2, pp. 155–185. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(01)00358-9. URL: https://www.sciencedirect.com/science/article/pii/S0304397501003589 (visited on 05/09/2023).

Bruijn, N. G. de (Jan. 1994). “A Survey of the Project Automath**Reprinted from: Seldin, J. P. and Hindley, J. R., eds., To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, p. 579-606, by courtesy of Academic Press Inc., Orlando.” en. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. Vol. 133. Selected Papers on Automath. Elsevier, pp. 141–161. DOI: 10.1016/S0049-237X(08)70203-9. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08702039 (visited on 07/20/2023).

Clavel, M. et al. (Aug. 2002). “Maude: specification and programming in rewriting logic”. en. In: *Theoretical Computer Science*. Rewriting Logic and its Applications 285.2, pp. 187–243. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(01)00359-0. URL: https://www.sciencedirect.com/science/article/pii/S0304397501003590 (visited on 05/09/2023).

Cockx, Jesper (2019). “Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules”. en. In.

Cockx, Jesper, Frank Piessens, and Dominique Devriese (2014). “Overlapping and Order-Independent Patterns”. en. In: *Programming Languages and Systems*. Ed. by Zhong Shao. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 87–106. ISBN: 978-3-642-54833-8. DOI: 10.1007/978-3-642-54833-8_6.

Cousineau, Denis and Gilles Dowek (2007). "Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo". en. In: *Typed Lambda Calculi and Applications*. Ed. by Simona Ronchi Della Rocca. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 102–117. ISBN: 978-3-540-73228-0. DOI: `10.1007/978-3-540-73228-0_9`.

Curry, H. B. (Nov. 1934). "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11, pp. 584–590. ISSN: 0027-8424. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1076489/` (visited on 07/21/2023).

Diaconescu, Răzvan and Kokichi Futatsugi (Aug. 2002). "Logical foundations of CafeOBJ". en. In: *Theoretical Computer Science*. Rewriting Logic and its Applications 285.2, pp. 289–318. ISSN: 0304-3975. DOI: `10.1016/S0304-3975(01)00361-9`. URL: `https://www.sciencedirect.com/science/article/pii/S0304397501003619` (visited on 05/09/2023).

Gonthier, Georges et al. (2013). "A Machine-Checked Proof of the Odd Order Theorem". en. In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 163–179. ISBN: 978-3-642-39634-2. DOI: `10.1007/978-3-642-39634-2_14`.

Harper, Robert, Furio Honsell, and Gordon Plotkin (Jan. 1993). "A framework for defining logics". en. In: *Journal of the ACM* 40.1, pp. 143–184. ISSN: 0004-5411, 1557-735X. DOI: `10.1145/138027.138060`. URL: `https://dl.acm.org/doi/10.1145/138027.138060` (visited on 03/23/2023).

Leino, K. Rustan M. (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". en. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 348–370. ISBN: 978-3-642-17511-4. DOI: `10.1007/978-3-642-17511-4_20`.

Martin-Löf, Per (Jan. 1975a). "About Models for Intuitionistic Type Theories and the Notion of Definitional Equality". en. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by Stig Kanger. Vol. 82. Proceedings of the Third Scandinavian Logic Symposium. Elsevier, pp. 81–109. DOI: `10.1016/S0049-237X(08)70727-4`. URL: `https://www.sciencedirect.com/science/article/pii/S0049237X08707274` (visited on 07/13/2023).

— (Jan. 1975b). "An Intuitionistic Theory of Types: Predicative Part". en. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by H. E. Rose and J. C. Shepherdson. Vol. 80. Logic Colloquium '73. Elsevier, pp. 73–118. DOI: `10.1016/S0049-237X(08)71945-1`. URL: `https://www.sciencedirect.com/science/article/pii/S0049237X08719451` (visited on 07/20/2023).

Peterson, Gerald E. and Mark E. Stickel (Apr. 1981). "Complete Sets of Reductions for Some Equational Theories". en. In: *Journal of the ACM* 28.2, pp. 233–264. ISSN: 0004-5411, 1557-735X. DOI: `10.1145/322248.322251`. URL: `https://dl.acm.org/doi/10.1145/322248.322251` (visited on 03/16/2023).

Rondon, Patrick M., Ming Kawaguci, and Ranjit Jhala (June 2008). "Liquid types". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. New York, NY, USA: Association for Computing Machinery, pp. 159–169. ISBN: 978-1-59593-860-2. DOI: `10.1145/1375581.1375602`. URL: `https://doi.org/10.1145/1375581.1375602` (visited on 07/20/2023).

Saillard, Ronan (Sept. 2015). "Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice". en. PhD thesis. Ecole Nationale Supérieure des Mines de Paris. URL: `https://pastel.archives-ouvertes.fr/tel-01299180` (visited on 05/08/2023).

Sjöberg, Vilhelm and Stephanie Weirich (Jan. 2015). "Programming up to Congruence". en. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai India: ACM, pp. 369–382. ISBN: 978-1-4503-3300-9. DOI: `10.1145/2676726.2676974`. URL: `https://dl.acm.org/doi/10.1145/2676726.2676974` (visited on 02/02/2023).

Spies, Simon and Yannick Forster (Jan. 2020). "Undecidability of higher-order unification formalised in Coq". en. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. New Orleans LA USA: ACM, pp. 143–157. ISBN: 978-1-

4503-7097-4. DOI: 10.1145/3372885.3373832. URL: https://dl.acm.org/doi/10.1145/3372885.3373832 (visited on 06/20/2023).

Swierstra, Wouter (July 2008). “Data types à la carte”. en. In: *Journal of Functional Programming* 18.4. Publisher: Cambridge University Press, pp. 423–436. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796808006758. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/data-types-a-la-carte/14416CB20C4637164EA9F77097909094 (visited on 06/20/2023).

Acronyms

AST	abstract syntax tree
AC	associative and commutative
LF	Edinburgh Logic Framework
LHS	left hand side
RHS	right hand side
WHNF	weak head normal form

Appendix A

Improved Injectivity Analysis

In section 4.5 we have described how we made the injectivity analysis work with rewrite rules. However, we also made the injectivity analysis stronger in general, as is described in this appendix.

For some operators like the minimum operator($\cap$), the injectivity analysis as described in section 4.5 is sufficient. As an example $suc\ \_x \cap \mathsf{suc}\ \_y \hookrightarrow \mathsf{suc}(\_x \cap \_y)$ is the only rule with suc as a possible RHS head. So, assuming we have not made $\cap$ definitionally AC, the old injectivity would be able to solve $?1 \cap ?2 = \mathsf{suc}\ ?3$. However, this simple algorithm fails when there are multiple rewrite rules with the correct RHS. This happens for example with addition, where both have suc as a possible RHS head:

$$\mathsf{suc}\ \_x + \_y \hookrightarrow \mathsf{suc}(\_x + \_y)$$
$$\_x + \mathsf{suc}\ \_y \hookrightarrow \mathsf{suc}(\_x + \_y)$$

The injectivity analysis would not be able to solve $a + ?1 = \mathsf{suc}\ a$, even though only one of the rules has a LHS that can match. We solve this limitation and more with the following enhancements:

- Rules are filtered based on both RHS and LHS, allowing some more cases of injectivity analysis on addition as described above.
- The most general rule is used when multiple overlapping rewrite rules apply. If no rule is most general, then injectivity analysis is stuck like before.

New Injectivity Analysis without Equal Heads

As a first step, we define which rules are potential candidates for reducing the blocked term. No rewrite rule can match directly, because then the term would not be blocked. Potential reducers are rewrite rules that are stuck on some meta-variable. This excludes rewrite rules that can never match, those would not be stuck and just fail. From the potential reducers, we can remove all rules that result in the wrong head symbol. Applying any of those would make type-checking fail. Note that the RHS head symbol might be refined by the rewrite rule matching. For example the rewrite rules $\_x + zero \hookrightarrow \_x$, will take its RHS head symbol from the LHS matching result.

Now that we have a list of rewrite rules that can potentially reduce the blocked term. We know that we must have at least one of them apply to make type-checking succeed. However, we do not want to make refinements that turn out to be wrong. Thus the only option here is to check if one of the LHSs is most general.

The implementation of this check instantiates every LHS with new variables and checks if one of the rules matches on every instantiated LHS. If there is a LHS that matches on all instantiated LHSs, then it is most general. The most general LHS can be used to refine the blocked term without losing any solution. To apply the refinement, each argument of the blocked term is conversion checked with the most general LHS.

If the blocked term has its arguments successfully refined, then one of the computation rules can now reduce the refined term. This means that it is now possible to retry conversion checking of the refined term with the WHNF term. Every step of injectivity does a reduction and typed-checked terms are finite in size, so this cycle is guaranteed to terminate.

Appendix B

Green Slime

Having a `UnSorted` list indexed by some `Bag` as defined in section 5.1, we might want to make use of our knowledge of items in the list. We would like to write a function `idx` that locates the position of an item that we know is in a list.

However, there is a phenomenon, which is sometimes called 'green slime'. This is when a data-structure has a function call as an index to a type constructor. And our `UnSorted` list data-structure is a good example:

```
data UnSorted : Bag → Set where
  [] : UnSorted ∅
  _::_ : (x : Nat) {xb : Bag} → UnSorted xb
      → UnSorted (bag x ⊔ xb)

idx1 : ∀ y {xb : Bag} → UnSorted (bag y ⊔ xb) → Nat
idx1 y (x :: xs) = ?
```

The _::_ constructor has a result type `UnSorted (bag x ⊔ xb)`, which is indexed by a call to the function _⊔_. This currently makes it impossible in Agda to pattern match on values of type `UnSorted b`, where `b` is not just a variable. The above code gives the following error message:

```
I'm not sure if there should be a case for the constructor _::_,
because I get stuck when trying to solve the following unification
problems (inferred index ≟ expected index):
  bag x ⊔ xb ≟ bag y ⊔ xb₁
when checking that the pattern x :: xs has type UnSorted (bag y ⊔ xb)
```

The reason for this is that pattern matching needs to be exhaustive, but also needs to make the argument definitionally equal to the pattern. To make the argument and the pattern equal, Agda uses a unifier. To make the unifier exhaustive, it can only unify function calls if the functions are guaranteed injective. Currently the only functions that Agda guarantees to be injective are constructors. It is thus impossible for this unifier to unify an arbitrary function application with something that is not just a variable.

I think it could be interesting if instead of substituting the variables, pattern matching just adds a new equality to the context. A type theory based on congruence closures (or equality reflection) as described by Sjöberg and Weirich (2015) would make these equalities definitional. As there is no need to find a substitution of the variables anymore, green slime could be solved completely. The downside is that beta-reduction can not be made definitional at the same time as congruence closures.

B.1 With-Abstraction and Instance Arguments

Despite the green slime, it is possible to get the `idx` function working in Agda, without any proof writing. First we need to move the offending propositional equality into a separate argument to make pattern matching possible:

```
idx2 : ∀ y {xb yb : Bag} → (xs : UnSorted xb) → (xb ≡ bag y ⊔ yb) → Nat
idx2 y (x :: xs) p = ?
```

By weakening the invariant from `xb ≡ bag y ⊔ yb` to `y ∈ xb`, we make the inductive step easier. This is where with-abstraction allows us to abstract over an expression such as $y == x$ and replace all occurrences by the possible values. This is enough to make the `idx` implementation work without any proof writing:

```
_||_ : Bool → Bool → Bool
false || x = x
true || x = true

postulate
  x||true : ∀ x → x || true ≡ true
  x||false : ∀ x → x || false ≡ x
  x==x : ∀ x → (x == x) ≡ true
{-# REWRITE x||true x||false x==x #-}

postulate
  _∈_ : Nat → Bag → Bool
  ∈-dist : ∀ x xs ys → x ∈ (xs ⊔ ys) ≡ (x ∈ xs) || (x ∈ ys)
  x∈∅ : ∀ x → x ∈ ∅ ≡ false
  x∈bag-y : ∀ x y → x ∈ bag y ≡ (x == y)
{-# REWRITE ∈-dist x∈∅ x∈bag-y #-}

idx : ∀ x {@0 xb : Bag} → (xs : UnSorted xb) → {{p : x ∈ xb ≡ true}} → Nat
idx y (x :: xs) with (y == x)
... | true = zero
... | false = suc (idx y xs)
```

By making the new requirement `y ∈ xb ≡ true` into an instance argument, it is passed automatically to the recursive call. The instance argument also makes it possible to call `idx` in any context where the requirement can be resolved with `refl`. Because of `x==x`, this is the case for the original function signature and we can thus write:

```
idx' : ∀ x {xb yb : Bag} → (xs : UnSorted (bag x ⊔ yb)) → Nat
idx' y xs = idx y xs
```

With-abstraction is not magic, the implementation needs to search for all occurrences of the abstracted expression and replace them by a new parameter. The above `idx` function gets desugared into something like the following:

```
mutual
  idx : ∀ y {xb : Bag} → (xs : UnSorted xb) → {{p : y ∈ xb ≡ true}} → Nat
  idx y (x :: xs) = idx-aux y x xs (y == x)

  idx-aux : (y x : Nat) {xb : Bag} (xs : UnSorted xb) (y==x : Bool)
    → {{p : y==x || (y ∈ xb) ≡ true}} → Nat
  idx-aux y x xs true = zero
  idx-aux y x xs false = suc (idx y xs)
```

Appendix C

Example Code

C.1 Sorting Algorithms

Two sorting functions are formally verified to preserve items and sort elements. This is done with an intrinsic approach as described in section 5.1.

```
{-# OPTIONS --rewriting --no-fast-reduce #-}

open import Agda.Builtin.List
open import Agda.Builtin.Nat
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite

case_of_ : ∀ {l₁ l₂} {A : Set l₁} {B : Set l₂} → A → (A → B) → B
case x of f = f x

data Rem : Set where
  just : Nat → Rem

unrem : Rem → Nat
unrem (just x) = x

unrem-eq : ∀ d → just (unrem d) ≡ d
unrem-eq (just x) = refl

{-# REWRITE unrem-eq #-}

infixr 22 _⊕_
_⊕_ : Nat → Rem → Nat
zero ⊕ just d = d
suc x ⊕ d = suc (x ⊕ d)
```

First we define a new operator on natural numbers, written ⊕. This operator behaves like addition, but we will not make it commutative. We make the second argument a newtype to enforce this.

```
data Compare : Nat → Nat → Set where
    ⩽ : ∀ {x d} → Compare x (x ⊕ d)
    ⩾ : ∀ {x d} → Compare (x ⊕ d) x

cmp : (x y : Nat) → Compare x y
cmp zero y = ⩽
cmp x zero = ⩾
cmp (suc x) (suc y) = case (cmp x y) of \where
    ⩽ → ⩽
    ⩾ → ⩾
```

Comparisons use Agdas limited pattern matching unification to refine numbers into applications of ⊕. Every number can only be refined once with this method, but that is enough to verify our sorting functions.

```
infixl 20 _⊔_
infixr 21 _↑_
postulate
  Bag : Set
  ∅ : Bag
  [zero] : Bag
  -- disjoin union: takes the sum of the number of elements in each bag
  _⊔_ : Bag → Bag → Bag
  -- add a number to all elements of the bag
  _↑_ : Nat → Bag → Bag

  ↑-comb : ∀ x xs ys → (x ↑ xs) ⊔ (x ↑ ys) ≡ x ↑ (xs ⊔ ys)
  ↑-dist : ∀ x y xs → (x ⊕ y) ↑ xs ≡ x ↑ (unrem y ↑ xs)
  x↑∅ : ∀ x → x ↑ ∅ ≡ ∅

  xs⊔∅ : ∀ xs → xs ⊔ ∅ ≡ xs
  ∅⊔xs : ∀ xs → ∅ ⊔ xs ≡ xs

  ⊔-comm : (xs ys : Bag) → xs ⊔ ys ≡ ys ⊔ xs
  ⊔-assoc : (xs ys zs : Bag) → (xs ⊔ ys) ⊔ zs ≡ xs ⊔ (ys ⊔ zs)

{-# REWRITE ∅⊔xs xs⊔∅ x↑∅ ↑-comb ↑-dist #-}

bag : Nat → Bag
bag x = x ↑ [zero]

{-# COMMASSOC ⊔-comm ⊔-assoc #-}
```

Unfortunately we can only postulate our `Bag` datastructure. This is because the normal form that we want can not be expressed with `data`. We want everything to reduce to applications of ↑, but this function is not injective.

```
infixr 5 _∷_
data UnSorted : @0 Bag → Set where
  [] : UnSorted ∅
  _∷_ : (x : Nat) {@0 xb : Bag} → UnSorted xb
      → UnSorted (bag x ⊔ xb)

data Sorted : @0 Bag → Set where
  [] : Sorted ∅
  _∷_ : (x : Nat) {@0 xb : Bag} → Sorted (x ↑ xb)
      → Sorted (bag x ⊔ x ↑ xb)
```

Our `Sorted` list data-structure refines the bag of the remainder. This sets a lower bound on the numbers in that bag and guarantees that the list is sorted.

```
insert : (x : Nat) {@0 yb : Bag} (ys : Sorted yb)
  → Sorted (bag x ⊔ yb)
insert x [] = x ∷ []
insert x (y ∷ ys) = case (cmp x y) of \where
  ⩽ → x ∷ y ∷ ys
  ⩾ → y ∷ insert x ys

insert-sort : {@0 xb : Bag} (xs : UnSorted xb) → Sorted xb
insert-sort [] = []
insert-sort (x ∷ xs) = insert x (insert-sort xs)
```

With everything in place, insertion sort is straightforward.

```
{-# TERMINATING #-}
merge : {@0 xb : Bag} (xs : Sorted xb) {@0 yb : Bag} (ys : Sorted yb)
  → Sorted (xb ⊔ yb)
merge [] ys = ys
merge xs [] = xs
merge xxs@(x ∷ xs) yys@(y ∷ ys) = case (cmp x y) of \where
        ⩽ → x ∷ merge xs yys
        ⩾ → y ∷ merge xxs ys

data Tree : @0 Bag → Set where
  empty : Tree ∅
  leaf : (x : Nat) → Tree (bag x)
  node : {@0 xb yb : Bag} → Tree xb → Tree yb → Tree (xb ⊔ yb)

insertT : {@0 xb : Bag} → (x : Nat) → Tree xb → Tree (bag x ⊔ xb)
insertT x empty = leaf x
insertT x (leaf y) = node (leaf x) (leaf y)
insertT x (node xs ys) = node ys (insertT x xs)

dealT : {@0 xb : Bag} → (xs : UnSorted xb) → Tree xb
dealT [] = empty
dealT (x ∷ xs) = insertT x (dealT xs)

mergeT : {@0 xb : Bag} → Tree xb → Sorted xb
mergeT empty = []
mergeT (leaf x) = x ∷ []
mergeT (node xs ys) = merge (mergeT xs) (mergeT ys)

merge-sort : {@0 xb : Bag} (xs : UnSorted xb) → Sorted xb
merge-sort xs = mergeT (dealT xs)
```

Our merge-sort works by first building a balanced tree structure and then merging branches. The tree allows structural recursion so that termination checking is easier Altenkirch, McBride, and McKinna 2005. Note that the 'merge' function had to be annotated with `{-# TERMINATING #-}`. This is because the termination checker gets confused by the variable refinements made by the pattern matching unifier. The error message mentions the two recursive calls as problematic:

```
merge xs (x ⊕ d ∷ ys)
merge (y ⊕ d ∷ xs) ys
```

C.2 Bag of Nats

This bag data-structure without postulates as described in section 5.2.2.

```
{-# OPTIONS --rewriting --no-fast-reduce #-}

open import Agda.Builtin.Nat
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite

cong : ∀ {A B : Set} (P : A → B) {x y : A}
  → x ≡ y → P x ≡ P y
cong _ refl = refl

-- this is always not empty
data Bag1 : Set where
  [zero] : Bag1
  zero : Bag1 → Bag1
  suc : Bag1 → Bag1

infixl 20 _⊔'_
_⊔'_ : Bag1 → Bag1 → Bag1
[zero] ⊔' ys = zero ys
zero xs ⊔' ys = zero (xs ⊔' ys)
xs ⊔' [zero] = zero xs
xs ⊔' zero ys = zero (xs ⊔' ys)
suc xs ⊔' suc ys = suc (xs ⊔' ys)

⊔'-[zero] : (xs : Bag1) → xs ⊔' [zero] ≡ zero xs
⊔'-[zero] [zero] = refl
⊔'-[zero] (zero xs) = cong zero (⊔'-[zero] xs)
⊔'-[zero] (suc xs) = refl

⊔'-zero : (xs ys : Bag1) → xs ⊔' (zero ys) ≡ zero (xs ⊔' ys)
⊔'-zero [zero] ys = refl
⊔'-zero (zero xs) ys = cong zero (⊔'-zero xs ys)
⊔'-zero (suc xs) ys = refl

{-# REWRITE ⊔'-[zero] ⊔'-zero #-}

⊔'-comm : (xs ys : Bag1) → xs ⊔' ys ≡ ys ⊔' xs
⊔'-comm [zero] ys = refl
⊔'-comm xs [zero] = refl
⊔'-comm xs (zero ys) = cong zero (⊔'-comm xs ys)
⊔'-comm (zero xs) ys = cong zero (⊔'-comm xs ys)
⊔'-comm (suc xs) (suc ys) = cong suc (⊔'-comm xs ys)

⊔'-assoc : (xs ys zs : Bag1) → (xs ⊔' ys) ⊔' zs ≡ xs ⊔' (ys ⊔' zs)
⊔'-assoc [zero] ys zs = refl
⊔'-assoc xs [zero] zs = refl
⊔'-assoc xs ys [zero] = refl
⊔'-assoc (zero xs) ys zs = cong zero (⊔'-assoc xs ys zs)
⊔'-assoc xs (zero ys) zs = cong zero (⊔'-assoc xs ys zs)
⊔'-assoc xs ys (zero zs) = cong zero (⊔'-assoc xs ys zs)
```

```
⊔'-assoc (suc xs) (suc ys) (suc zs) = cong suc (⊔'-assoc xs ys zs)

{-# COMMASSOC ⊔'-comm ⊔'-assoc #-}

bag1 : Nat → Bag1
bag1 zero = [zero]
bag1 (suc x) = suc (bag1 x)

-- this bag can be empty
data Bag : Set where
  ∅ : Bag
  just : Bag1 → Bag

infixl 20 _⊔_
_⊔_ : Bag → Bag → Bag
∅ ⊔ ys = ys
xs ⊔ ∅ = xs
just xs ⊔ just ys = just (xs ⊔' ys)

⊔-∅ : (xs : Bag) → xs ⊔ ∅ ≡ xs
⊔-∅ ∅ = refl
⊔-∅ (just x) = refl

{-# REWRITE ⊔-∅ #-}

⊔-comm : (xs ys : Bag) → xs ⊔ ys ≡ ys ⊔ xs
⊔-comm ∅ ys = refl
⊔-comm xs ∅ = refl
⊔-comm (just xs) (just ys) = refl

⊔-assoc : (xs ys zs : Bag) → (xs ⊔ ys) ⊔ zs ≡ xs ⊔ (ys ⊔ zs)
⊔-assoc ∅ ys zs = refl
⊔-assoc xs ∅ zs = refl
⊔-assoc xs ys ∅ = refl
⊔-assoc (just xs) (just ys) (just zs) = refl

{-# COMMASSOC ⊔-comm ⊔-assoc #-}

bag : Nat → Bag
bag x = just (bag1 x)
```

C.3 Effect Rows

This is the full code for the effect example from section 5.3.

```
{-# OPTIONS --rewriting --no-fast-reduce #-}

open import Agda.Builtin.Nat
open import Agda.Builtin.Bool
open import Agda.Builtin.Maybe
open import Agda.Builtin.Unit
open import Agda.Builtin.List
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
```

```
cong : ∀ { a b} { A : Set a } { B : Set b }
       (f : A → B ) {m n} → m ≡ n → f m ≡ f n
cong f refl = refl

case_of_ : ∀ {l₁ l₂} {A : Set l₁} {B : Set l₂} → A → (A → B) → B
case x of f = f x

cast : ∀ {l m} {A : Set l} (P : A → Set m)
    {x y : A} → x ≡ y → P x → P y
cast _ refl x = x

data Eff : Set where
    Bools : Eff
    Nats : Eff

get-eff : Eff → Set
get-eff Bools = List Bool
get-eff Nats = List Nat

cmp : (x y : Eff) → Maybe (x ≡ y)
cmp Nats Nats = just refl
cmp Bools Bools = just refl
cmp _ _ = nothing

record Monoid (A : Set) : Set where
  field
    mempty : A
    _<>_ : A → A → A
open Monoid {{...}}

infixl 20 _&&'_
_&&'_ : Bool → Bool → Bool
true &&' x = x
false &&' _ = false

x&&'true : ∀ x → x &&' true ≡ x
x&&'true false = refl
x&&'true true = refl

&&'false : ∀ x → x &&' false ≡ false
&&'false false = refl
&&'false true = refl

x&&'x : ∀ x → x &&' x ≡ x
x&&'x false = refl
x&&'x true = refl

{-# REWRITE x&&'true &&'false x&&'x #-}

&&'-comm : ∀ (x y : Bool) → x &&' y ≡ y &&' x
&&'-comm false y = refl
&&'-comm true y = refl

&&'-assoc : ∀ (x y z : Bool) → (x &&' y) &&' z ≡ x &&' (y &&' z)
```

```
&&'-assoc false y z = refl
&&'-assoc x false z = refl
&&'-assoc x y false = refl
&&'-assoc true true true = refl

{-# COMMASSOC &&'-comm &&'-assoc #-}

Cond : Set
Cond = List Eff → Bool

infixl 20 _&&_
_&&_ : Cond → Cond → Cond
p && q = λ x → (p x) &&' (q x)

_++_ : {A : Set} → List A → List A → List A
[] ++ r = r
(x :: l) ++ r = x :: (l ++ r)

xs++[] : ∀ {A} (xs : List A) → xs ++ [] ≡ xs
xs++[] [] = refl
xs++[] (x :: xs) = cong (_::_ x) (xs++[] xs)

{-# REWRITE xs++[] #-}


data WithEff : ∀ {L : List Eff} (v : Bool) (O : Set) → Set₁ where
    pure : ∀ {v O} → O → WithEff {[]} v O
    emit : ∀ {L E v} → get-eff E → WithEff {E :: L} v ⊤
    inc : ∀ {L E v O} → WithEff {L} v O → WithEff {E :: L} v O
    _>>=_ : ∀ {L v X O} → WithEff {L} v X → (X → WithEff {L} v O) → WithEff {L} v O
    fail : ∀ {L O} → WithEff {L} false O

_>>_ : ∀ {L v O} → WithEff {L} v ⊤ → WithEff {L} v O → WithEff {L} v O
left >> right = left >>= λ tt → right

WithEff' : (c : Cond) (O : Set) → Set₁
WithEff' c O = {xs : List Eff} → WithEff {xs} (c xs) O

raise : ∀ {B v L O} → WithEff {L} true O → WithEff {B ++ L} v O
raise {[]} {false} s = fail
raise {[]} {true} s = s
raise {E :: L} {v} s = inc (raise {L} {v} s)

smart-pure : ∀ {B v O} → O → WithEff {B} v O
smart-pure {B} {v} o = raise {B} {v} (pure o)

data Index (x : Eff) : (List Eff) → Set where
    here : ∀ {xs} → Index x (x :: xs)
    there : ∀ {y xs} → Index x xs → Index x (y :: xs)
    nvm : Index x []

idx : ∀ x xs → Index x xs
idx y [] = nvm
idx y (x :: xs) = case (cmp x y) of λ where
    (just refl) → here
    nothing → there (idx y xs)

eff-idx : ∀ {x xs} → Index x xs → Bool
eff-idx here = true
eff-idx (there i) = eff-idx i
eff-idx nvm = false

eff : (E : Eff) → Cond
eff E = λ xs → eff-idx (idx E xs)

emit-idx : {v : Bool} → ∀ {x xs} → (i : Index x xs) → get-eff x
    → WithEff {xs} (v &&' eff-idx i) ⊤
emit-idx here e = emit e
emit-idx {v} (there i) e = inc (emit-idx {v} i e)
emit-idx nvm e = fail

smart-emit : ∀ {E v xs} → {{v &&' eff E xs ≡ v}} → get-eff E → WithEff {xs} v ⊤
smart-emit {E} {v} {xs} {{p}} e = cast (λ x → WithEff x ⊤) p (emit-idx {v} (idx E xs) e)

smart-raise : ∀ {L v w O} → {{v &&' w ≡ v}} → WithEff {L} w O → WithEff {L} v O
smart-raise (pure x) = pure x
smart-raise (emit x) = emit x
smart-raise (inc xs) = inc (smart-raise xs)
smart-raise (xs >>= f) = (smart-raise xs) >>= (λ x → smart-raise (f x))
smart-raise {{refl}} fail = fail

data Handle (O E : Set) : Set where
    _,_ : O → E → Handle O E

handle : ∀ {e L O v} → {{f : Monoid (get-eff e)}} → WithEff {e :: L} v O
    → WithEff {L} v (Handle O (get-eff e))
handle fail = fail
handle (emit e) = smart-pure (tt , e)
handle (inc s) = do
    r ← s
    smart-pure (r , mempty)
handle (s1 >>= s2) = do
    (l , e1) ← handle s1
    (r , e2) ← handle (s2 l)
    smart-pure (r , (e1 <> e2))

eval : ∀ {O} → WithEff {[]} true O → O
eval (pure x) = x
eval (s1 >>= s2) = eval (s2 (eval s1))

instance
  ListMonoid : ∀ {A} → Monoid (List A)
  mempty {{ListMonoid}} = []
  _<>_ {{ListMonoid}} xs ys = xs ++ ys

test : WithEff' (eff Nats && eff Bools) ⊤
test = do
    smart-emit {Nats} (10 :: [])
    smart-emit {Bools} (true :: [])
    smart-emit {Nats} (42 :: [])

main : List Nat
main = eval do
    (nats , bools) ← handle {Bools} do
      (tt , nats) ← handle {Nats} test
      smart-pure nats
    smart-pure nats

main1 : WithEff' (eff Bools) (List Nat)
main1 = do
    (tt , nats) ← handle {Nats} test
    smart-pure nats

test2 : WithEff' (eff Nats && eff Bools) ⊤
test2 = do
    _ ← smart-raise main1
    smart-pure tt
```