Term Indexed Tries

Victor Cacciari Miraldo and Wouter Swierstra

University of Utrecht

Abstract. Pen-and-paper mathematics is said to enjoy the advantage of being *simple*. One can just write equations underneath equation, as long as we know a given equivalence between sub-equations hold. Computer systems tipically do enjoy that degree of elegance. One is forced to provide tons of purely mechanical code to acomplish the simplest of tasks. On this paper we address this problem, using Agda as our target. We describe the high-level of a generic rewrite engine for Agda, using meta-programming techniques to generate these mechanical pieces of code. We also tackle a generalization towards a fully automatic *auto* tactic in Agda.

1 Introduction

On [Mir15] we explore the construction of a rewriting engine for Agda [Nor07]. We use generic programming techniques, and Agda's support for such, to generate the terms that justify the given rewrite steps. The users needs only to provide the name of the lemma they want to apply. We are going to illustrate our framework in section 3. Agda's Reflection module allows the programmer to access the meta-representation of terms during typechecking. They can also generate other terms and plug them back in, mid-compilation.

A reader already familiar with Agda might argue that small rewrites are quite simple to perform using the *rewrite* keyword. When the need to perform equational reasoning over complex formulas arises, however, we still need to specify the substitutions manually in order to apply a theorem to a subterm. This manual specification is error prone and purely mechanical, as we shall see in this paper.

Not to mention the innability to use the *rewrite* keyword with relations other than propositional equality. Sometimes, it might be interesting to perform equational reasoning over other domains. Relational Algebra is an interesting example, as relational equality is a tricky concept to encode in the Intensional Theory of Types [Mir15].

Even though our framework is capable of figuring out which parameters to apply to the user supplied lemma, we want to go a step further. What if our framework could find which lemma to apply, given a database of lemmas? The idea for Term-indexed Tries arises from the need to make such lemma-database efficient, at least for look up, given that String-indexed Tries are the data-structure of choice for searching substrings.

Working with strings as indexes for a trie, however, is very easy (as their functor is linear on the recursive arguments). We will see how things get a bit trickier once we start working with generic functors. Another stone in the way is how should we handle variables. Theorems and lemmas, especially when defined on proof-assistants, assume a general form, for instance, right identity for addition states that for all x, $x + 0 = x$. We can instantiate x to anything here. Why does right identity works for justifying that $\pi \times (r^2 + 0) = \pi \times r^2$? Because there is an instantiation, namelly $x \mapsto r^2$, which when applied to the lemma, with congruence $\pi \times \Box$, closes the proof.

After having a solid rewrite engine, our objective became coding this structure for storing lemmas in Agda, and release it as part of the rewriting framework of [Mir15]. Agda is a pure, functional language with a dependent type system built on top of the Theory of Types, Martin Löf, [ML84]. We refer the reader to [NPS90,Nor09] for a good introduction on both the theory and practice of Agda. Performance problems stoped the development of our Term-Trie in Agda, specially within the insertion function. We still need time to properly pinpoint the problem. Taking into account, also, that it is more likely that readers are familiar with Haskell, we are going to present the datastructures and algorithms we use in Haskell.

The *agda-rw* library is publicaly available on the internet in the following GitHub repository.

https://github.com/VictorCMiraldo/agda-rw

2 Basic Agda

In languages such as Haskell or ML, where a Hindley-Milner based algorithm is used for type-checking, values and types are clearly separated. Values are the objects being computed and types are simply tags to *categorize* them. In Agda, however, this story changes. There is no distinction between types and values, which gives a whole new level of expressiveness to the programmer.

The Agda language is based on the intensional theory of types by Martin-Löf [ML84]. Great advantages arise from doing so, more specifically in the programs-as-proofs mindset. Within the Intensional Theory of Types, we gain the ability to formally state the meaning of quantifiers in a type. We refer the interested reader to [NPS90].

Datatype definitions in Agda resemble Haskell's GADTs syntax [VWPJ06]. Let us illustrate the language by defining fixed-size Vectors. For this, we need natural numbers and a notion of sum first.

```
data ℕ : Set where            _+_ : ℕ → ℕ → ℕ
  Z : ℕ                       Z + n = n
  S : ℕ → ℕ                   (S m) + n = S (m + n)
```

The $\mathbb{N}$: *Set* statement is read as *Nat is of kind* $*$, for the Haskell enthusiast. In a nutshell, *Set*, or, Set_0, is the first universe, the type of small types. For consistency reasons, Agda has an infinite number of non-cumulative universes inside one another. That is, $\text{Set}_i \nsubseteq \text{Set}_i$ but $\text{Set}_i \subseteq \text{Set}_{i+1}$.

The underscores in the definition of + indicate where each parameter goes. This is how we define mix-fix operators. We can use underscores virtually anywhere in a declaration, as long as the number of underscores coincide with the number of parameters the function expects.

3 Why building a new tactic?

In order to illustrate the application of the *agda-rw* library, let us consider the proof of commutativity, over sum, for natural numbers. The proof itself is very simple, and as we will witness, two simple lemmas and a induction hypothesis suffice to complete the proof.

```
succomm : ∀ m n → m + suc n ≡ suc (m + n)
identity : ∀ n → n + 0 ≡ n

comm : ∀ m n → m + n ≡ n + m
comm zero n = sym (identity n)
comm (suc m) n =
  begin
    suc m + n
  ≡⟨ refl ⟩
    suc (m + n)
  ≡⟨ cong suc (comm m n) ⟩*
    suc (n + m)
  ≡⟨ sym (succomm n m) ⟩
    n + suc m
  □
```

Fig. 1. Addition commutativity over Natural Numbers

Here ≡ is Agda Propositional Equality type. Whenever we say $x \equiv y$ we mean that x and y *evaluate to the same value*. Let us take a more delicate look on what is going on on the above proof. The first step, $(sucm) + n \equiv suc(m+n)$ is justified by the definition of + in the standard library, therefore they must evaluate to the same value. The second justification, however, is a congruence. We can think of this congruence as some sort of context for rewriting. Note how the rewrite happens inside the *suc* context. The recursive call models the induction hypothesis. The last step is pretty straight forward. Here, *sym* denotes symmetry of ≡.

Note the step marked with a $\cdot^\star$. The term *cong suc* (*comm m n*) is exactly what is generated by calling our tactic *by*, passing *comm* as the lemma we want to apply. In the rest of this document we will explain how did we accomplish this.

Rewriting in Agda

During any kind of mathematical reasoning, in a pen-and-paper context, one usually uses a fair amount of implicit rewrites. Yet, we cannot skip these steps in a proof assistant. We need to really convince Agda that two things are equal, by Agda's equality notion, before it can syntatically rewrite the terms.

Let us see the definition of Propositional Equality, in Agda. Remembering that $x \equiv y$ means that x and y evaluate to the same value. This is enforced by repeating the variable in the type of *refl*.

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

Having a proof $p : x \equiv y$ convinces Agda that x and y will *evaluate* to the same value. Whenever this is the case, we can rewrite x for y in a predicate. The canonical way to do so is using the *subst* function:

```
subst : {A : Set} (P : A → Set) {x y : A} → x ≡ y → P x → P y
subst P refl p = p
```

Here, the predicate P can be seen as a context where the rewrite will happen. From a programming point of view, Agda's equality notion makes perfect sense! Yet, whenever we are working with more abstract concepts, we might need a finer notion of equality. However, this new equality must agree with Agda's equality if we wish to perform syntactical rewrites. As we will see in the next section, this is not always the case.

It is worth mentioning a subtle detail on the definition of *subst*. Note that, on the left hand side, the pattern p has type $P\ x$, according to the type signature. Still, Agda accepts this same p to finish up the proof of $P\ y$. What happens here is that upon pattern matching on *refl*, Agda knows that x and y evaluate to the same value. Therefore it basically substitutes, in the current goal, every y for x. As we can see here, pattern-matching in Agda actually allows it to infer additional information during type-checking.

4 Reflection in Agda

As of version 2.4.8, Agda's reflection API provides a few keywords displayed in table 1. This feature can the thought of as the Template Haskell approach in Agda, or, meta programming in Agda.

The idea is to access the abstract representation of a term, during compile time, perform computations over it and return the resulting term before resuming compilation. We will not delve into much detail on Agda's abstract representation. The interested reader should go to Paul van der Walt's thesis[vdW12], where, although somewhat outdated, Paul gives a in-depth explanation of reflection in Agda. However, the following excerpt gives a taste of how reflection looks like.

```
example : quoteTerm (λ x → suc x) ≡ con (quote suc) []
example = refl
```

Although very small, it states that the abstract representation of suc is a constructor, whose name is suc and has no arguments whatsoever. The *Term* datatype, exported from the Reflection module, has a large number of constructors and options. We are just interested in a small subset of such terms, which is enough motive to build our own term datatype. That is the reason why we will not explain reflection in depth. Nevertheless, the next section provides a discussion on our interface to reflection.

Another interesting factor is to note how Agda normalizes a term before quoting it. Note how it performed a η-reduction automatically. To prevent this is the reason why we need all those *record* boilerplate in our Relational Algebra library.

quote	Returns a *Name* datatype, representing the quoted identifier.
quoteTerm	Takes a term, normalizes it and returns it's *Term* inhabitant.
quoteGoal g *in* f	Brings a quoted version of the goal in place into *f*'s scope, namely, *g*.
quoteContext	Returns a list of quoted types. This is the ordered list of types of the local variables from where the function was called.
tactic f	Is syntax-sugar for *quoteGoal g in* (*f quoteContext g*).

Table 1. Agda 2.4.8 Reflection API

The general idea behind *agda-rw* is to use Agda's Reflection API to generate the congruences and substitutions needed to justify the rewrites we seek to perform, such as the marked congruence in figure 1. In the following sections we will give a brief explanation of the operations we need to generate such terms.

4.1 Terms and Rewriting

Before some confusion arises it is important to explain that we can not tackle rewriting in a standard fashion, like arbitrary term rewriting systems might do. Our problem is, in fact, slightly different. Given two structurally different terms and an action, we want to see: (A) if it is possible to apply such action in such a way that (B) it justifies the rewriting step. The first step is a simple matter of unification, which will be discussed later. However, the second part, justifying the rewrite, boils down to the generation of a subst that explains to Agda that what we are doing is valid. Our main goal here is to generate such subst.

In the Reflection module, Agda provides us with the term-language they use. It is worth mentioning that their definition is over the top for what we need.

We will handle terms only. No need to worry about half the constructors they provide. For this reason, we decided to make our own AST. This can also be helpful on the future, if Agda changes its Reflection interface.

A Term is something in the standard lambda-calculus format. However, we split the base case in three separate cases. An *ovar* is an outer variable. This provides a way for us to plug in different types inside a Term. The *ivar* construct represents a DeBruijn indexed variable. Literals are pretty self-explanatory.

```
data RTerm {a} (A : Set a) : Set a where
  ovar : (x : A) → RTerm A
  ivar : (n : ℕ) → RTerm A
  rlit : (l : Literal) → RTerm A
  rlam : RTerm A → RTerm A
  rapp : (n : RTermName) (ts : List (RTerm A)) → RTerm A
```

Applications, however, need to be distinguished between constructor applications and definitions. We also add *impl* to represent the arrow type _ → _.

```
data RTermName : Set where
  rcon : Name → RTermName
  rdef : Name → RTermName
  impl : RTermName
```

One of the advantages of having a parametric Term datatype is the guarantees that we can provide with it. Specially using a dependent type system. We can represent closed terms, $RTerm\ \bot$, where $\bot$ represents the empty type. We can represent terms with holes using $RTerm\ (Maybe\ A)$. We can use the parameter to plug in additional information needed for specific situations, for instance, finite indexes for unification, as in [McB03].

We will not delve into the technicalities of converting Agda terms to *RTerm*s. The technique we used was inspired by *Auto in Agda* [KS15], cf. [Mir15]. Given our term AST, we will now illustrate the important operations for: (A) inferring the context of the rewrite and (B) infering the parameters to pass to the user-supplied lemma.

– *maybe introduce lists in Agda? I mean, they are the same as in Haskell...*

Assume one is trying to prove associativity of list concatenation. Here, _ ++ _ denotes the canonical definition of concatenation and $x :: xs$ is the list with x as its head and xs as its tail.

```
assoc : ∀ {xs ys zs} → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
assoc nil _ _ = refl
assoc (x :: xs) ys zs = ?
```

The obvious solution to the hole above is *cong* (_ :: _ *x*) (*assoc xs ys zs*). Basically, the *action* that fits in the problem is *assoc*, which corresponds to the induction hypothesis. The rest of the information can be inferred, automatically, from the goal and action type, whose types are given in figure **??**.

$$
\begin{array}{ll}
\text{Goal} & \underbrace{(x \; :: \; xs \; +\!\!+ \; ys) \; +\!\!+ \; zs}_{g_1} \overset{hd_g}{\equiv} \underbrace{x \; :: \; xs \; +\!\!+ \; (ys \; +\!\!+ zs)}_{g_2} \\
\text{Action} & \quad \underbrace{(xs \; +\!\!+ \; ys) \; +\!\!+ \; zs}_{a_1} \overset{hd_a}{\equiv} \underbrace{xs \; +\!\!+ \; (ys \; +\!\!+ zs)}_{a_2}
\end{array}
$$

Fig. 2. Goal and Action for hole zero

Looking at both terms, it is not hard to infer that it is some sort of term-intersection that is going to give the abstraction (context) to use it as a congruence. If we use strucutral intersection, though, we have $g_\square = g_1 \cap g_2 = (x :: \square +\!\!+ \square)$, where $\square$ represents a hole, or the part in which the terms differ. We obtain a term with two holes! We should lift definitions whose arguments are all holes, to a single hole.

$$g_\square = g_1 \cap g_2 = (x \; :: \; \square \; +\!\!+ \; \square) \overset{lift}{\to} (x \; :: \; \square)$$

Term intersection and lifting are easily defined by recursion on terms. The type signature for the aforementioned operations are as follows.

```
_ ∩ _ : {A : Set} {{eqA : Eq A}}
  → RTerm A → RTerm A → RTerm (Maybe A)
_ ↑ : ∀ {a} {A : Set a} → RTerm (Maybe A) → RTerm (Maybe A)
```

Here, we already have ways of infering the context under which the rewrite happens. We also have the lemma that is going to be applied. Now, it is a matter of figuring out the parameters that need to be supplied to the lemma.

The context, $g_\square$, can be seen as a road-map. It makes sense to be able to compute the difference of a given term and $g_\square$. That is, $g_\square - g_2 \equiv xs +\!\!+ (ys +\!\!+ zs)$. This operation is defined by recursion on both terms. While we keep seeing equal constructors, we keep traversing. When we find the hole, we return the second term. If a diffence is encountered, we return *nothing*.

Let us denote the type of our lemma, *assoc*, using De Bruijn indexes. We have $t = \lambda\lambda\lambda.(0 +\!\!+ 1) +\!\!+ 2 \equiv 0 +\!\!+ (1 +\!\!+ 2)$. A simple instantiation of $g_\square - g_i$ with t_i yields $\{0 \mapsto xs, 1 \mapsto ys, 2 \mapsto zs\}$. Which is sufficient to generate the term that justifies the rewrite. The substitution gives the parameters, in order, that should be applied to the lemma and the term $g_\square$ gives the context in which the rewrite happens.

4.2 Library Structure

Up to this point, we gave a small description of how we accomplish rewrites in Agda. Yet, different domains might require different terms. For it is also important to outline the structure of the library and its respective API.

One interesting aspect of Agda is the possibility to have parametric modules, that is, modules that receive parameters when they are imported.

The top level module is `RW.RW`, and this module receives a list of *Term Strategies*, or something of type *TStratDB*. This strategy record specifies both a predicate to indicate when this strategy should be used, given the goal and lemma topmost relations, and how to construct the final term given the information computed internally.

Below we find, verbatim, the strategy provided for reasoning over propositional equality.

```
module RW.Strategy.PropEq where
  pattern patEq = (rdef (quote _===_))
  private
    open UData
    when : RTermName → RTermName → Bool
    when patEq patEq = true
    when _ _ = false
    fixTrs : Trs → RTerm  → RTerm
    fixTrs Symmetry term = rapp (rdef (quote sym)) (term :: [])
    how : Name → UData → Err StratErr (RTerm )
    how act (u − data gSq s trs)
      = i₂ (rapp (rdef (quote cong))
                 (hole2Abs gSq
                  :: foldr fixTrs (makeApp act s) trs
                  :: [])
           )
  propeqStrat : TStrat
  propeqStrat = record
    {when = when
    ;how = how
    }
```

The constructors *rapp* and *rdef* works for building function applications and specifying definitions, just like *app* and *def* in `Reflection`. The *r* prefix stands for our own term datatype. This allows a easier integration with future, possibly incompatible, versions of the Reflection module.

The *when* predicate specifies that when both topmost relations are the propositional euality we should use *how* to compute the final term. Whereas *how* uses the information provided by the backend, which comes through in a record, and generates the correct congruence. We transform the hole of *gSq*, which represents $g_\square$, into an abstraction, we append a call to *symmetry* when needed and we apply the substitution's terms to the action supplied by the user. This is returned as a closed term which is later on plugged back in by Agda's reflection mechanism.

In order to use the *RW* library with this strategy, one should import it as follows.

open import *RW.Strategy.PropEq* **using** (*propeq* − *strat*)
open import *RW.RW* (*propeq* − *strat* :: [])

Note that *RW.RW* receives a list of strategies, so one can use rewriting for different relations in the same equational reasoning proof.

5 Summary and Generalizations

We exposed the problem of rewriting in Agda and gave a description of how to use the metaprogramming features at hand to approach a solution. The solution described is in fact what is implemented and released as stable in our repository[1].

Regardless, we explored how to perform one rewrite given one action. It is natural, hence, to ask how could one perform many rewrites given one action (for each rewrite) or one rewrite given a large choice of actions. It is clearly unfeasible to have many rewrites with many actions, as this would be a fully automatic theorem prover.

We choose to tackle the second option; of one rewrite given an action database. The reason for this is the absence of run-time type information for many identifiers, leading to a state-explosion for intermediate states of the *many rewrites*. A broader explanation is found on [Mir15].

6 Term-Indexed Tries

In this section we will tackle the *one rewrite, many actions* generalization of our rewrite tactic. The underlying idea is to design some structure that can store terms such that looking up is better than a list lookup. We fetch inspiration on String-indexed tries, and on it's datatype generic cousin [HJL04]. Our lookup routine, however, is slightly more complicated, as it has to deal with variable instantiation on the fly.

Following the popular saying – A picture is worth a thousand words – let us begin with a simple representation. Just like a trie, our RTrie trie was designed to store names of actions to be performed, indexed by their respective type. Figure 4 shows the RTrie that stores the actions from figure 3. For clarity, we have written the De Bruijn indexes of the respective variables as a superscript on their names.

Let us illustrate the lookup of, for instance, $(2 \times x) + 0 \equiv 2 \times x$. The search starts by searching in the root's partial map for $\equiv$. We're given two tries! Since

[1] https://github.com/VictorCMiraldo/agda-rw

$$x^0 + 0 \equiv x^0$$
$$x^0 + y^1 \equiv y^1 + x^0$$
$$x^2 + (y^1 + z^0) \equiv (x^2 + y^1) + z^0$$

Fig. 3. Identity, Commutativity and Associativity for addition.

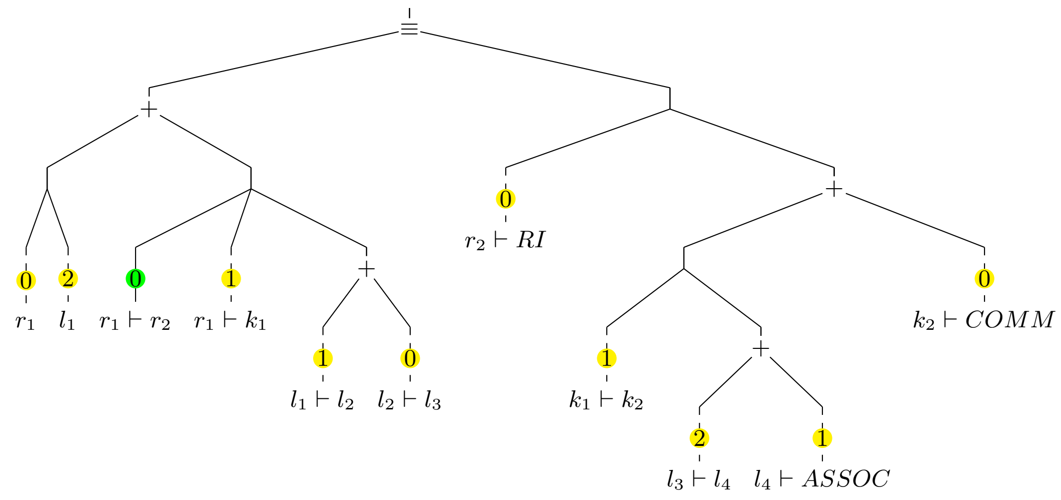

Fig. 4. RTrie for terms of figure 3. Yellow and green circles represent DeBruijn indexes and literals, respectively.

$\equiv^{\star_1}$ is a binary constructor. We proceed by looking for $(2 \times x) + 0$ in the left child of $\equiv$. Well, our topmost operator is now a $+^{\star_2}$, we repeat the same idea and now, look for $(2 \times x)$ in the left child of the left $+$. Here, we can choose to instantiate variable 0 as $(2 \times x)$ and collect label r_1 or instantiate variable 2, with the same term, and collect label l_1. Since we cannot know beforehand which variable to instantiate, we instantiate both! At this point, the state of our lookup is $(0 \mapsto 2 \times x, r_1) \vee (2 \mapsto 2 \times x, l_1)$. We proceed to look for the literal 0 in the right trie of $+^{\star_2}$, taking us to a leaf node with a rewrite rule stating $r_1 \vdash r_2$. This reads r_1 should be rewritten by r_2. We apply this to all states we have so far. Those that are not labeled r_1 are pruned. However, we could also instantiate variable 1 at that node, so we add a new state $(0 \mapsto 2 \times x, 1 \mapsto 0, k_1)$. At this step, our state becomes $(0 \mapsto 2 \times x, r_2) \vee (0 \mapsto 2 \times x, 1 \mapsto 0, k_1)$.

We go up one level and find that now, we should look for $2 \times x$ at the right child of $\equiv^{\star_1}$. We cannot traverse the right child labeled with a $+$, leaving us with to compare the instantiation gathered for variable 0 in the left-hand-side of $\equiv^{\star_1}$ to $2 \times x$. They are indeed the same, which allows us to apply the rule $r_2 \vdash RI$. Which concludes the search, rewriting label r_2 by RI, or, any other code for

+-right-identity. By returning not only the final label, but also the environment gathered, we get the instantiation for variables for free. The result of such search should be $(0 \mapsto 2 \times x, RI) :: []$.

This small worked example already provides a few insights not only on how to code lookup, but also on how to define our RTrie. We have faced two kinds of nodes. Fork nodes, which are composed by a list of cells, and, Leaf nodes, which contain a list of (rewrite) rules. Everything happens inside a non-deterministic state monad, in which each state has an enviroment for variable substitution and a label, indicating where we last performed our rewrite.

– *Now a "lookup" and "insert" section followed by a conclusion and we should be ok.*

References

[HJL04] Ralf Hinze, Johan Jeuring, and Andres Lh. Type-indexed data types. In *Science of Computer Programming*, pages 148–174, 2004.

[KS15] Pepijn Kokke and Wouter Swierstra. Auto in agda. In Ralf Hinze and Janis Voigtlnder, editors, *Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 276–301. Springer International Publishing, 2015.

[McB03] Conor McBride. First-order unification by structural recursion. *Journal of functional programming*, 13(06):1061–1075, 2003.

[Mir15] Victor Cacciari Miraldo. Proofs by rewriting in Agda. Master's thesis, Utrecht University and University of Minho, 2015. Submitted.

[ML84] P. Martin-Löf. Intuitionistic type theory, 1984.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Nor09] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[vdW12] Paul van der Walt. Reflection in Agda. Master's thesis, Utrecht University, the Netherlands, 2012.

[VWPJ06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. *SIGPLAN Not.*, 41(9):251–262, September 2006.