FORMALISING GHC'S TYPE SYSTEM

A RIGOROUS AND MACHINE CHECKED FORMULATION OF OUTSIDEIN(X) IN A DEPENDENTLY TYPED PROGRAMMING LANGUAGE

A THESIS BY

LIAM O'CONNOR-DAVIS

SUPERVISED BY

MANUEL M. T. CHAKRAVARTY

SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE HONOURS DEGREE OF
Bachelor of Science (Computer Science)

School of Computer Science and Engineering
University of New South Wales

THE UNIVERSITY OF
NEW SOUTH WALES

SCIENTIA

MANU ET MENTE

SYDNEY · AUSTRALIA

OCTOBER 16TH 2012

Abstract

GHC, a state of the art Haskell compiler (Marlow (Ed.), 2010), offers numerous extensions to the standard Haskell type system (Schrijvers *et al.*, 2009; Yorgey *et al.*, 2012; Kiselyov *et al.*, 2010; Peyton Jones *et al.*, 2007). Each of these extensions is usually specified only semi-formally, and only in isolation. Very little work has been done examining type system properties when multiple type system extensions are combined, which is the scenario actually being faced by GHC developers. To address this, the GHC team published OutsideIn(X), a mostly-rigorous formulation of GHC's type inference system (Vytiniotis *et al.*, 2011), which encompasses every type system extension developed for GHC to date.

We formalise OutsideIn(X) in a mechanical proof assistant, in order to provide a body of formal work upon which future extensions can be developed. By using a mechanical proof assistant we not only ensure correctness of our proofs and complete rigour in our definitions, but also make possible the incremental development of the formal work alongside the more practically-minded type checker implementation in GHC. This additional accessibility will hopefully prevent further extensions from being developed without regard to the effect such an extension may have on other parts of the type system.

Our formalisation is developed in Agda (Norell, 2008). As a dependently typed programming language which enforces totality, Agda doubles as a proof assistant (Howard, 1980). It is still under heavy development, and is quite experimental. By formalising OutsideIn(X) in Agda, we demonstrate its readiness for type system work, and also provide an example to encourage further type systems research in Agda.

Contents

“There are many ways of trying to understand programs. People often rely too much on one way, which is called ‘debugging’ and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.”

— Robin Milner, *Foreword to The Little MLer*

1 Introduction

Haskell is a purely functional programming language, with a type system that supports algebraic data types, type inference, parametric (higher-kinded) polymorphism, and type class constraints (Marlow (Ed.), 2010). In recent years, the developers of GHC, a prominent Haskell compiler, have implemented a variety of extensions to this type system, with the aim of providing greater expressiveness, ease of use, or static verification capabilities. Some are straightforward, such as generalising type classes to type relations via multi-parameter type classes. Some require more significant extensions to the type system, such as type families (Kiselyov *et al.*, 2010) and the earlier functional dependencies extension; some make type inference significantly more difficult and are major extensions, such as GADTs (Schrijvers *et al.*, 2009), impredicative polymorphism and arbitrary-rank types (Peyton Jones *et al.*, 2007).

While GHC accommodates all of these extensions simultaneously, the papers that introduce each one discuss type inference and type checking only in isolation, and sometimes quite informally. This makes the properties of GHC's type reconstruction algorithm difficult to determine when multiple extensions are combined.

As a first step towards solving this problem, the GHC team (specifically Vytiniotis, Peyton Jones, Schrijvers and Suzmann) published OutsideIn(X), a modular type inference system that accommodates all of these extensions (and possibly more), along with soundness and principality proofs (Vytiniotis *et al.*, 2011).

Our work aims to more rigorously formalise OutsideIn(X) in the dependently-typed programming language *cum* proof assistant Agda 2 (Norell, 2008).

1.1 Related Work

The OutsideIn(X) system itself is similar in presentation to the HM(X) system, a parameterised formalisation of ML's type inference, first presented in (Odersky *et al.*, 1997) and more rigorously formalised in (Pottier & Rémy, 2005). It is the culmination of years of work and is the latest in a series of type inference systems, starting with the original inference system for GADTs in Haskell based on "wobbly types" (Schrijvers *et al.*, 2009), right up to the very similar LHM(X) system presented in (Vytiniotis *et al.*, 2010).

Very little work has been done on formalising type inference in a proof assistant, and what little work that has been done is primarily focused on type inference for ML. Nipkow and Narachewski have formalised Milner's original $\mathcal{W}$ algorithm for the HM calculus in Isabelle/HOL and proven soundness (Naraschewski & Nipkow, 1999), and Dubois et al. simultaneously performed a similar verification in Coq (Dubois & Ménissier-Morain, 1999) for the purposes of developing a certified ML compiler (Dubois, 2000). Surprisingly, no type inference algorithms have been formalised in Agda before. Even more surprisingly, no algorithm which includes support for GADTs or other advanced type system features has ever been formalised in a proof assistant, to our knowledge.

The techniques we use for term representation are drawn heavily from Bird's observation of monadic structure in syntax trees (Bird & Paterson, 1999), a concept first explored in (Bellegard & Hook, 1994). We also draw several representational tricks from works in generic programming using dependent types (Morris *et al.*, 2004) and McBride's work on structurally recursive unification (McBride, 2003), which is used directly in our simple instantiation.

1.2 Why mechanise **OutsideIn**(X)*?*

Any handwritten formalisation or proof will likely lack the amount of rigour necessary to be accepted by a proof assistant, much like a handwritten algorithm is unlikely to be accepted by a programming language compiler. Formalising OutsideIn(X) *in a proof assistant* is therefore more difficult than it appears at first glance. The use of a proof assistant requires us to redesign those parts of the system that are not amenable to automated checking, and to make rigorous all those parts of the original formalisation that are left to the reader's intuition.

By formalising OutsideIn(X) in a proof assistant, we achieve two main goals. Firstly, we make explicit that which was implicit, and to prove that which was assumed in the original OutsideIn(X) paper, ensuring that our formal work stands on solid ground; and secondly, we encourage those developing extensions for the type system to use our work as a foundation for their formalisation, along with the necessary practical implementation of the new extension in GHC's type checker. By making our formalisation available as code, we hope to mitigate the social problem of formal work on a type system being published in a long paper and subsequently ignored[1].

1.3 Why Agda?

Agda is an interesting choice of proof assistant for this task. This choice was made not simply because Agda is the most familiar to the author, but also because Agda is still quite experimental, and the subject of a great deal of new research. By formalising OutsideIn(X) in Agda, we show Agda's readiness for type systems work, and provide an example for others researching type systems and considering Agda. Similar work has been done in Coq (Dubois & Ménissier-Morain, 1999) and Isabelle (Naraschewski & Nipkow, 1999), but our formalisation is the first such work in Agda.

1.4 A Brief Introduction to Agda

Agda is a programming language with a concrete syntax similar to Haskell, based on the dependent intuitionistic type theory of Per Martin-Löf (Martin-Löf, 1984). Agda enforces totality by mandating that all functions be structurally recursive[2], meaning that programs correspond to proofs in a higher order intuitionistic logic. Features include (co)-inductive data types and families, "mix-fix" syntax (Danielsson & Norell, 2011), parameterised modules, "View from the left" style pattern matching (McBride & McKinna, 2004) and compile time proof irrelevance annotations. For a complete tutorial in Agda programming, we defer to the experts (Norell, 2008); the Agda examples in this thesis only require a rudimentary knowledge of Agda's syntax.

1.4.1 A Key Point of Difference

Unlike other dependently typed theorem provers such as Coq, when working in Agda one does not write a proof script consisting of a series of proof *tactics* which transform or generate a proof *object* (i.e. the dependently typed program); the program or proof is written directly. This has two main implications:

1. *Proof terms are explicit, automation is not available.* There is limited opportunity for automated generation and manipulation of the proof object (i.e. complicated proof tactics) when one writes the proof object code directly. Recently, a new reflection interface was added in Agda version 2.3.0, which allows Agda programs to inspect the current goal and generate solutions for it. It offers a kind of automatic generation of proof objects using Agda itself as a tactic language, however it remains highly experimental and few tactics are available. Our formalisation does not make use of this feature.
2. *Great care must be taken to keep representations manageable.* In most theorem provers, the formal properties we want to prove and the definitions they describe tend to be quite distinct. Because Agda uses the same language to talk about both, we can combine them in ways that would be impossible in Isabelle/HOL or unusual in Coq[3]. Key properties about our definitions are implied by their structure, rather than independently proven as lemmas, using a variety of definitional tricks. We employ these tricks extensively for our representation of names, substitutions, and abstract syntax trees.

[1] See the new `DataKinds` extension (Yorgey *et al.*, 2012)
[2] Or, in the case of coinduction, structurally corecursive.
[3] Such tricks are certainly possible in Coq, but less commonly employed.

1.5 Overview

There are three main components to our work, each discussed in a separate chapter:

1. Our approach to OutsideIn(X) is discussed in chapter two, where we rework various parts of OutsideIn(X) that were informally presented originally, providing a more rigorous formalisation.
2. Our encoding of the system in Agda, including our method of term representation, is explained in chapter three. As discussed in the previous chapter, the choice of representation is very important for Agda work.
3. Chapter four introduces a simple instantiation of the X parameter, which allows us not only to sanity-check our definitions, but also to provide a testbed for experimentation with the system.

Lastly, chapter five discusses future directions for this work, including proof work and instantiating the system for Haskell itself.

2 Making OutsideIn(X) rigorous

OutsideIn(X), published in (Vytiniotis *et al.*, 2011), is an approach to type inference approach that supports modular type inference and, most interestingly, *local assumptions*, such as those introduced by pattern matching on a *generalised algebraic data type*, or GADT. A GADT is more general than a regular algebraic data type because its constructors can have substantially more flexible types, including constraints and type variables not mentioned in the constructed type. This allows for a variety of useful features, such as existential quantification and type indexing (Schrijvers *et al.*, 2009).

For example, suppose we had the following GADT.

$$\begin{array}{lll} \textbf{data} & \mathit{EqOrShow} :: * \to * \ \textbf{where} & \\ & \text{IsEq} \quad :: & (\mathit{Eq}\ \tau) \Rightarrow \mathit{EqOrShow}\ \tau \\ & \text{IsShow} \quad :: & (\mathit{Show}\ \tau) \Rightarrow \mathit{EqOrShow}\ \tau \end{array}$$

When we pattern match on a GADT such as this, we introduce a *local assumption*, in this case about the type variable α:

$$\begin{array}{l} f :: \mathit{EqOrShow}\ \alpha \to \alpha \to \mathit{Either}\ \mathit{String}\ \mathit{Bool} \\ f\ \text{IsEq}\ x = \text{Right}\ (x \equiv x) \\ f\ \text{IsShow}\ x = \text{Left}\ (\mathit{show}\ x) \end{array}$$

Here, the local assumption $\mathit{Eq}\ \alpha$ allows the first alternative to type check, while the local assumption $\mathit{Show}\ \alpha$ allows the second alternative to type check. It is important to note that these assumptions must *remain local.* This is especially apparent when the constraints are contradictory, for example using equality constraints[4]:

$$\begin{array}{lll} \textbf{data} & \mathit{IntOrBool} :: * \to * \ \textbf{where} & \\ & \text{IsInt} \quad :: & (\tau \sim \mathit{Int}) \Rightarrow \mathit{IntOrBool}\ \tau \\ & \text{IsBool} \quad :: & (\tau \sim \mathit{Bool}) \Rightarrow \mathit{IntOrBool}\ \tau \end{array}$$

A function which pattern matches on this GADT must obviously localise each constraint assumption to each alternative — it is impossible for $\alpha \sim \mathit{Bool}$ and $\alpha \sim \mathit{Int}$ unless $\mathit{Int} \sim \mathit{Bool}$, which is of course not the case[5].

These local assumptions have historically been difficult to deal with, resulting in lack of principal types. While the general typing rules in a language may allow terms which lack principal types, OutsideIn(X) only infers principal types. If a term lacks a principal type, then type inference will fail — it is not complete. Most similar systems, for example HM(X), are complete but lack support for GADTs and type classes (Pottier & Rémy, 2005).

2.1 Regarding `let`*-generalisation*

When it was first published, OutsideIn(X) was not the inference system used by GHC. Some significant changes had to be made to the static semantics of GHC Haskell in order to accommodate it; specifically, generalisation of inferred types in local `let`-expressions was removed.

[4] More conventionally, these constructors would not use type variables at all, instead each constructor would be parameterised by the concrete types *Int* and *Bool* respectively, however this is desugared into a type variable with an explicit equality constraint (Schrijvers *et al.*, 2009)

[5] At least, it is not the case in languages outside those whose names contain the letter C and not much else.

Continuing from our earlier example, if we had the following definition:

$$
\begin{array}{l}
f :: \mathit{IntOrBool}\ \alpha \rightarrow \alpha \rightarrow \mathit{Bool} \\
f\ x\ y = \textbf{let}\ g\ z\ = \mathit{not}\ y\ \textbf{in} \\
\qquad\qquad \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\quad \begin{array}{lcl} \mathit{IsInt} & \rightarrow & \mathrm{True} \\ \mathit{IsBool} & \rightarrow & g\ () \end{array}
\end{array}
$$

Most programmers would expect the binding g to give a type error, as it requires y to be of type *Bool*, *before* that has been established via pattern matching on x[6]. A principal type does exist, however, for g:

$$g :: \forall\beta.(\alpha \sim \mathit{Bool}) \Rightarrow \beta \rightarrow \mathit{Bool}$$

That is, the constraint $\alpha \sim \mathit{Bool}$, rather than being rejected, is *abstracted over* in the inferred type. Then, at any call site, we must provide evidence that $\alpha \sim \mathit{Bool}$, which can be done in this example thanks to the pattern matching.

The authors of OutsideIn(X) state that abstracting over all inferred constraints imposes a significant complexity cost on the implementation of the type checker, for two main reasons. Firstly, at each call site of such a generalised expression, the (potentially large) set of constraints that have been abstracted over must be shown satisfiable, which, if necessary for every locally bound expression, would be difficult to perform efficiently.

Secondly, it becomes impossible to solve constraints "on-the-fly" in a similar manner to Milner's $\mathcal{W}$ algorithm (Milner, 1978); instead the compiler must generate all constraints and then solve them in discrete phases. GHC relies on this on-the-fly solving to resolve equality constraints efficiently using mutable type variables (Peyton Jones *et al.*, 2007). They also observe that the principal types inferred by such a method are often highly confusing and result in the type checker accepting almost-certainly erroneous code, such as the above example.

In OutsideIn(X), this generalisation and abstraction of inferred constraints is only performed *at the top level*, i.e. where the type environment is empty. Therefore, the difficulties above disappear. At the local level, *no generalisation is performed at all.* That is, an (unannotated) `let`-binding **let** $x = y$ **in** e is equivalent to $(\lambda x.\ e)\ y$. While a cherished feature of most type inference algorithms since the original $\mathcal{W}$, `let` generalisation at a local level turns out to be relied upon very rarely in practice, with a total of only 127 lines needing to be modified in the `base` Haskell library consisting of 94,954 lines after this change was made (Vytiniotis *et al.*, 2010).

2.2 Phases of Inference

As mentioned in the previous section, GHC resolves equality constraints as soon as they can be solved for efficiency reasons. Specifically, only those constraints which rely on information not available when they are generated are deferred until after constraint generation for solving. All other constraints are solved immediately. OutsideIn(X), however, presents inference as two separate phases:

1. *Generate* constraints (according to a set of syntax-directed rules), combining them into one large overall constraint.
2. *Solve* the constraint, using unification and standard constraint-solving methods.

The method used by GHC then can be viewed then as interleaving these two phases, whereas OutsideIn(X) (at least in theory) keeps them distinct. Our formal work is derived from OutsideIn(X), not GHC, and for such formal work we do not particularly care about the *efficiency* of type inference, but rather the various properties we can prove about it. For this reason our formalisation also separates these phases. Indeed, we add *additional* phases to this process in order to more clearly separate the solver phase; as well as to deal with our extended constraint language, discussed in the next section.

[6] Note that $\mathit{not} :: \mathit{Bool} \rightarrow \mathit{Bool}$

Term variables		$\in$	x, y, z, f, g, h
Type variables		$\in$	a, b, c
Unification type variables		$\in$	$\alpha, \beta, \gamma, \delta$
Data constructors		$\in$	K
Type constructors		$\in$	$\mathtt{T}$
	υ	$::=$	$K \mid x$
Expressions	e	$::=$	$\upsilon \mid \lambda x.\ e \mid e_1\ e_2$
		$\mid$	$\mathtt{case}\ e\ \mathtt{of}\ \{\overline{K\ \bar{x} \to e}\}$
		$\mid$	$\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \mathtt{let}\ x :: \sigma = e_1\ \mathtt{in}\ e_2$
Type schemes	σ	$::=$	$\forall \bar{a}.\ Q \Rightarrow \tau$
Programs	$prog$	$::=$	$f :: \sigma = e, prog \mid f = e, prog \mid \epsilon$
Constraints*	Q	$::=$	$\epsilon \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid \cdots$
Extended constraints	C	$::=$	$Q \mid C_1 \wedge' C_2 \mid \exists \bar{\beta}.\ Q \supset C \mid \exists \alpha.\ C$
Monotypes*	τ	$::=$	$tv \mid \mathtt{T}\ \bar{\tau} \mid \tau_1 \to \tau_2 \mid \cdots$
	tv	$::=$	a
Environments	Γ	$::=$	$\epsilon \mid (\upsilon : \sigma), \Gamma$
Axiom Schema*	$\mathcal{Q}$	$::=$	$\cdots$

Γ_0 : Types of data constructors*
$K : \forall \bar{a}\bar{b}.\ Q \Rightarrow \bar{\tau} \to \mathtt{T}\ \bar{a}$

$*$ — part of the parameter X

Fig. 1: Our updated syntax, slightly extended from OutsideIn(X)

2.3 The Extended Constraint Language

Constraint generation in OutsideIn(X) is specified independently of the exact constraint system or type system used — the constraint and type terms are part of the X parameter. Clearly, some intuitive conditions must be met by these components, summarised in two additional parameters: an *entailment relation* of global constraint schema to locally inferred constraints, and certain *simplifier conditions* that ensure that the provided solver behaves consistently with this entailment relation. Specifically, the constraint system must include constraint conjunction and equality constraints (see Fig. 1) that behave as one would expect, and any locally inferred constraints must be resolved trivially if they restate an axiom in a global constraint scheme (see Fig. 2). This parameterisation of the system is similar to HM(X), the parameterised extension of ML's type inference presented in (Odersky *et al.*, 1997)[7], with the addition of global *axiom schema*, which are designed to accommodate Haskell's type classes. Specifically, type class instances can generate top-level implication constraints such as $\mathit{Show}\ \alpha \Rightarrow \mathit{Show}\ [\alpha]$. In OutsideIn($X$), these top-level constraints are expressed in axiom schema and do not form part of the main constraint language.

In order to deal with local assumptions, OutsideIn(X) extends Q, the original constraint language in X, to an *algorithmic* constraint language C. C is just Q with an additional form, $\exists \bar{\beta}.\ (Q \supset C)$, where Q is a local assumption, C is a constraint, and $\bar{\beta}$ are the *only* variables that can be unified while solving the constraint $Q \supset C$. The local assumption is defined as a constraint in the *original* constraint language Q specifically, rather than the larger C, as all local assumptions come from the constraint clause of a generalised type signature, either provided by the user or in the type of a generalised data constructor; they are not generated locally by the algorithm itself.

Our presentation of this extended language differs from the presentation in OutsideIn(X) in several respects. One minor change is that we make the separation between the languages C, Q and $\mathcal{Q}$ much more clear. In particular, we add a new form of conjunction to the extended language C, written $\phi \wedge' \psi$, as the conjunction inherited from Q can, technically, only contain Q-constraints. In addition, we have changed $\mathcal{Q}$, the language of axiom schema, so that it no longer includes all of Q. Instead, we make no assumptions about the forms that $\mathcal{Q}$ schema may take, and have reformulated the entailment relation to require *both* a $\mathcal{Q}$-constraint *and* a Q-constraint as context (see Fig. 2). We have also added to the entailment relation requirements for conjunction elimination rules as well as a rule to deal with ϵ-constraints. These rules were inexplicably absent from the original presentation, despite being implicitly used repeatedly in the soundness proof presented in the same work.

[7] or the more rigorous formalisation presented by Pottier and Remy in their chapter of *Advanced Types and Programming Languages* (Pottier & Rémy, 2005)

$$\boxed{\mathcal{Q}; Q \Vdash Q}$$

part of the parameter X, subject to the following requirements:

Tautology	$\mathcal{Q}; Q \Vdash \epsilon$	(R_1)
Reflexivity	$\mathcal{Q}; Q \Vdash Q$	(R_2)
Transitivity	$\mathcal{Q}; Q_1 \Vdash Q_2$ and $\mathcal{Q}; Q_2 \Vdash Q_3$ implies $\mathcal{Q}; Q_1 \Vdash Q_3$	(R_3)
Substitution	$\mathcal{Q}; Q_1 \Vdash Q_2$ implies $\theta\mathcal{Q}; \theta Q_1 \Vdash \theta Q_2$ where θ is a type substitution	(R_4)
Conjunction intro.	$\mathcal{Q}; Q \Vdash Q_1$ and $\mathcal{Q}; Q \Vdash Q_2$ implies $\mathcal{Q}; Q \Vdash Q_1 \wedge Q_2$	(R_5)
Conjunction elim.	$\mathcal{Q}; Q_1 \wedge Q_2 \Vdash Q_1$	(R_6)
	$\mathcal{Q}; Q_1 \wedge Q_2 \Vdash Q_2$	(R_7)
Type eq. reflexivity	$\mathcal{Q}; Q \Vdash \tau \sim \tau$	(R_8)
Type eq. symmetry	$\mathcal{Q}; Q \Vdash \tau_1 \sim \tau_2$ implies $\mathcal{Q}; Q \Vdash \tau_2 \sim \tau_1$	(R_9)
Type eq. transitivity	$\mathcal{Q}; Q \Vdash \tau_1 \sim \tau_2$ and $\mathcal{Q}; Q \Vdash \tau_2 \sim \tau_3$ implies $\mathcal{Q}; Q \Vdash \tau_1 \sim \tau_3$	(R_{10})
Type eq. substitutivity	$\mathcal{Q}; Q \Vdash \tau_1 \sim \tau_2$ implies $\mathcal{Q}; Q \Vdash [\alpha \mapsto \tau_1]\tau \sim [\alpha \mapsto \tau_2]\tau$	(R_{11})

$$\boxed{\mathcal{Q}; Q_{\text{given}}; \overline{\alpha_{\text{tch}}} \overset{\text{simp}}{\mapsto} Q_{\text{wanted}} \rightsquigarrow Q_{\text{residual}}; \theta}$$

part of the parameter X, subject to the following requirements:

Touchable-aware:	$dom(\theta) \subseteq \overline{\alpha_{\text{tch}}}$
Soundness:	$\mathcal{Q}; Q_g \overset{\text{simp}}{\mapsto} Q_w \rightsquigarrow Q_r; \theta$ implies $\mathcal{Q}; Q_g \wedge Q_r \Vdash \theta Q_w$
Principality: (Guess-freedom)	$\mathcal{Q}; Q_g \overset{\text{simp}}{\mapsto} Q_w \rightsquigarrow Q_r; \theta$ implies $\mathcal{Q}; Q_g \wedge Q_w \Vdash Q_r \wedge \mathcal{E}_\theta$ where $\mathcal{E}_\theta = \{(\alpha \sim \tau) \mid [\alpha \mapsto \tau] \in \theta\}$.

Fig. 2: Entailment relation and simplifier conditions

The constraint solver is also part of the parameter X and therefore can only act on constraints in Q, not the extended constraint language C. Therefore, OutsideIn(X) includes additional machinery to solve implication constraints given a solver for Q-constraints. In OutsideIn(X), and in this thesis, the term *simplifier* is used to describe the Q-solver, whereas the term *solver* is reserved for the top-level C-solver.

2.4 Fresh Variables

A common sin against mathematical rigour often committed in type inference literature is that of the magically fresh variable. This is an example taken from a constraint generation rule (for lambda abstractions) in OutsideIn(X):

$$\frac{\alpha\ \textbf{fresh} \qquad \Gamma, (x : \alpha) \mapsto e : \tau \rightsquigarrow C}{\Gamma \mapsto \lambda x.\ e : (\alpha \to \tau) \rightsquigarrow C}$$

These fresh variables must be globally unique and in scope throughout the entire program, despite being summoned *ad-hoc* as constraints are generated; they must be completely unused variable names before being introduced here. Narachewski and Nipkow's approach to this problem, when they verified $\mathcal{W}$ in Isabelle, was to thread an infinite source of known globally unique variable names (i.e. a natural number n for which all names in $\{N_i | i \geqslant n\}$ are unique and unused) as state through the program, removing a name from the source when a fresh variable was introduced (i.e. incrementing n) (Naraschewski & Nipkow, 1999). While this approach is perhaps closer to how $\mathcal{W}$ would be implemented in a compiler, it has a certain inelegance that complicates Agda definitions of these rules considerably. Specifically, the rules would need to live within a state monad, introducing needless dependency between rule invocations which would otherwise be independent.

Our approach is instead to reuse some machinery that is already present in OutsideIn(X) for local assumptions. We shall extend the constraint language slightly while generating constraints, and simplify it again before solving them. Specifically, we add another form to the extended constraint language C: an (existential) quantifier for unification variables, which we denote with $\Finv\alpha.\ C$ ($\Finv$ is used here rather than $\exists$ to distinguish between the two existential quantifiers in C; $\exists$ is for local assumptions, and $\Finv$ is for unification variables). This approach is similar to the existential quantifiers used in (Pottier & Rémy, 2005).

To use these quantifiers, we must first rearrange the constraint generation rules so that the type is viewed as

$$\boxed{\Delta; \Gamma \mapsto e : \tau \rightsquigarrow C}$$

$$\frac{(v : \forall \bar{a}.\ Q_1 \Rightarrow \tau_1) \in \Gamma}{\Delta; \Gamma \mapsto v : \tau \rightsquigarrow \exists \bar{\alpha}.\ [\overline{a \mapsto \alpha}]Q_1 \wedge' (\tau \sim [\overline{a \mapsto \alpha}]\tau_1)}\ \textsc{VarCon}$$

$$\frac{\alpha_1, \alpha_2, \alpha_3, \Delta; \Gamma \mapsto e_1 : \alpha_1 \rightsquigarrow C_1 \qquad \alpha_1, \alpha_2, \alpha_3, \Delta; \Gamma \mapsto e_2 : \alpha_2 \rightsquigarrow C_2}{\Delta; \Gamma \mapsto e_1\ e_2 : \tau \rightsquigarrow \exists \alpha_1.\ \exists \alpha_2.\ \exists \alpha_3.\ C_1 \wedge' C_2 \wedge' (\alpha_1 \sim (\alpha_2 \to \alpha_3)) \wedge' (\tau \sim \alpha_3)}\ \textsc{App}$$

$$\frac{\alpha, \beta, \Delta; \Gamma, (x : \alpha) \mapsto e : \beta \rightsquigarrow C}{\Delta; \Gamma \mapsto \lambda x.\ e : \tau \rightsquigarrow \exists \alpha.\ \exists \beta.\ C \wedge' (\tau \sim (\alpha \to \beta))}\ \textsc{Abs}$$

$$\frac{\alpha_1, \alpha_2, \Delta; \Gamma \mapsto e_1 : \alpha_1 \rightsquigarrow C_1 \qquad \alpha_1, \alpha_2, \Delta; \Gamma, (x : \alpha_1) \mapsto e_2 : \alpha_2 \rightsquigarrow C_2}{\Delta; \Gamma \mapsto \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau \rightsquigarrow \exists \alpha_1.\ \exists \alpha_2.\ C_1 \wedge' C_2 \wedge' (\tau \sim \alpha_2)}\ \textsc{Let}$$

$$\frac{\alpha_1, \alpha_2, \Delta; \Gamma \mapsto e_1 : \alpha_1 \rightsquigarrow C_1 \qquad \alpha_1, \alpha_2, \Delta; \Gamma, (x : \alpha_1) \mapsto e_2 : \alpha_2 \rightsquigarrow C_2}{\Delta; \Gamma \mapsto \texttt{let}\ x :: \tau' = e_1\ \texttt{in}\ e_2 : \tau \rightsquigarrow \exists \alpha_1.\ \exists \alpha_2.\ C_1 \wedge' C_2 \wedge' (\tau \sim \alpha_2) \wedge' (\alpha_1 \sim \tau')}\ \textsc{LetA}$$

$$\frac{\begin{array}{c}\sigma_1 = \forall \bar{a}.\ Q \Rightarrow \tau' \qquad Q \neq \epsilon \text{ or } \bar{a} \neq \epsilon \qquad \bar{\alpha}, \beta_1, \beta_2, \Delta; \Gamma \mapsto e_1 : \beta_1 \rightsquigarrow C \\ C_1 = \exists \bar{\alpha}.\ \exists \epsilon.\ ([\overline{a \mapsto \alpha}]Q \supset C \wedge' \beta_1 \sim [\overline{a \mapsto \alpha}]\tau') \qquad \beta_1, \beta_2, \Delta; \Gamma, (x : \sigma_1) \mapsto e_2 : \beta_2 \rightsquigarrow C_2\end{array}}{\Delta; \Gamma \mapsto \texttt{let}\ x :: \sigma_1 = e_1\ \texttt{in}\ e_2 : \tau \rightsquigarrow \exists \beta_1.\ \exists \beta_2.\ C_1 \wedge' C_2 \wedge' (\tau \sim \beta_2)}\ \textsc{GLetA}$$

$$\frac{\begin{array}{c}\alpha, \beta, \bar{\gamma}, \bar{\delta}, \Delta; \Gamma \mapsto e : \alpha \rightsquigarrow C \\ (K_i : \forall \bar{a}\bar{b}.\ Q_i \Rightarrow \bar{\tau}_i \to \texttt{T}\ \bar{a}) \in \Gamma \qquad \alpha, \beta, \bar{\gamma}, \bar{\delta}, \bar{\rho}, \Delta; \Gamma, (\overline{x_i : [b \mapsto \rho][a \mapsto \gamma]\tau_i}) \mapsto e_i : \delta_i \rightsquigarrow C_i \\ C_i' = \begin{cases} C_i \wedge' \delta_i \sim \beta & \text{if } \bar{b}_i = \epsilon \text{ and } Q_i = \epsilon \\ \exists \bar{\rho}.\ \exists \epsilon.([\overline{b \mapsto \rho}][\overline{a \mapsto \gamma}]Q_i) \supset C_i \wedge' \delta_i \sim \beta & \text{otherwise} \end{cases}\end{array}}{\Delta; \Gamma \mapsto \texttt{case}\ e\ \texttt{of}\ \{\overline{K_i\ \bar{x}_i \to e_i}\} : \tau \rightsquigarrow \exists \alpha. \exists \beta. \exists \bar{\gamma}. \exists \bar{\delta}.\ C \wedge' (\texttt{T}\ \bar{\gamma} \sim \alpha) \wedge' (\bigwedge C_i') \wedge' (\tau \sim \beta)}\ \textsc{Case}$$

Fig. 3: Constraint Generation Rules (using our new quantifier)

input, rather than output. This does not affect the algorithm significantly — types are always just a single metavariable[8], and the exact form of the inferred type is instead indicated by an explicit equality constraint. This means that the only place where these new unification variables are mentioned is within the generated constraint, and not within the type:

$$\frac{\alpha\ \textbf{fresh} \qquad \beta\ \textbf{fresh} \qquad \Gamma, (x : \alpha) \mapsto e : \beta \rightsquigarrow C}{\Gamma \mapsto \lambda x.\ e : \tau \rightsquigarrow C \wedge (\tau \sim (\alpha \to \beta))}$$

Then, we can simply add an environment of available type variables Δ to the constraint generation judgement, and replace the fresh variable introductions with quantifiers (where α and β are not in the environment Δ):

$$\frac{\alpha, \beta, \Delta; \Gamma, (x : \alpha) \mapsto e : \beta \rightsquigarrow C}{\Delta; \Gamma \mapsto \lambda x.\ e : \tau \rightsquigarrow \exists \alpha.\ \exists \beta.\ C \wedge (\tau \sim (\alpha \to \beta))}$$

The changes required to most of the other rules are in a similar vein (See Fig. 3 for a full set). This more rigorous formulation of constraint generation is equivalent to the original presentation, with the added benefit of a formalized notion of fresh variables, and less ambiguity in the presentation of the constraint languages.

[8] This does not lead to ambiguity dangers, as the system is still syntax-directed

$$\boxed{C \overset{\text{pnx}_1}{\mapsto} C}$$

$$\begin{array}{lcll}
x \wedge' (\exists\alpha.\ y) & \overset{\text{pnx}_1}{\mapsto} & \exists\alpha.\ x \wedge' y & \textsc{Prenex}_1 \\
(\exists\alpha.\ x) \wedge' y & \overset{\text{pnx}_1}{\mapsto} & \exists\alpha.\ x \wedge' y & \textsc{Prenex}_2 \\
\exists\bar{\beta}.\ Q \supset (\exists\alpha.\ x) & \overset{\text{pnx}_1}{\mapsto} & \exists\bar{\beta}\alpha.\ Q \supset x & \textsc{Prenex}_3
\end{array}$$

$$\boxed{C \overset{\text{pnx*}}{\mapsto} C}$$

$$\frac{}{C \overset{\text{pnx*}}{\mapsto} C}\ \textsc{Refl*} \qquad \frac{C_1 \overset{\text{pnx}_1}{\mapsto} C_2 \qquad C_2 \overset{\text{pnx*}}{\mapsto} C_3}{C_1 \overset{\text{pnx*}}{\mapsto} C_3}\ \textsc{Trans*}$$

We say $C \overset{\text{pnx}}{\mapsto} C'$ iff $C \overset{\text{pnx*}}{\mapsto} C'$ and there exists no C'' such that $C' \overset{\text{pnx}_1}{\mapsto} C''$

Fig. 4: Prenexer rewrite rules, and their reflexive transitive closure

2.5 The Prenexer

Of particular interest is the interaction between local assumption forms and these fresh variable quantifiers. Here is a rule from the original OutsideIn(X) formulation where a user has specified a general type for a local `let` binding:

$$\frac{\begin{array}{c}\sigma_1 = \forall\bar{a}.\ Q_1 \Rightarrow \tau_1 \qquad Q_1 \neq \epsilon \text{ or } \bar{a} \neq \epsilon \qquad \Gamma \mapsto e_1 : \tau \rightsquigarrow C \qquad \bar{\beta} = \mathit{fuv}(\tau, C) - \mathit{fuv}(\Gamma) \\ C_1 = \exists\bar{\beta}.\ (Q_1 \supset C \wedge \tau \sim \tau_1) \qquad \Gamma, (x : \sigma_1) \mapsto e_2 : \tau_2 \rightsquigarrow C_2\end{array}}{\Gamma \mapsto \texttt{let}\ x :: \sigma_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \textsc{Original}$$

This rule introduces a local assumption constraint where the variables bound by the quantifier ($\exists\bar{\beta}$) are all free unification variables introduced when generating the constraint for e_1 and τ. With our new quantifiers, there will never be any free unification variables under any circumstances (as any unification variables introduced would have been bound within the constraint C), which makes the updated rule look somewhat odd — the existential quantifier in the local assumption form is empty[9]:

$$\frac{\begin{array}{c}\sigma_1 = \forall\bar{a}.\ Q \Rightarrow \tau' \qquad Q \neq \epsilon \text{ or } \bar{a} \neq \epsilon \qquad \bar{\alpha}, \beta_1, \beta_2, \Delta; \Gamma \mapsto e_1 : \beta_1 \rightsquigarrow C \\ C_1 = \exists\bar{\alpha}.\ \exists\epsilon.\ ([\overline{a \mapsto \alpha}]Q \supset C \wedge \beta_1 \sim [\overline{a \mapsto \alpha}]\tau') \qquad \beta_1, \beta_2, \Delta; \Gamma, (x : \sigma_1) \mapsto e_2 : \beta_2 \rightsquigarrow C_2\end{array}}{\Delta; \Gamma \mapsto \texttt{let}\ x :: \sigma_1 = e_1\ \texttt{in}\ e_2 : \tau \rightsquigarrow \exists\beta_1.\ \exists\beta_2.\ C_1 \wedge C_2 \wedge (\tau \sim \beta_2)}\ \textsc{GLetA}$$

We resolve this oddity by introducing a new constraint simplification phase, which is run before solving, called the *prenexer*. The prenexer is responsible for eliminating all of the new $\exists$-quantifiers introduced by constraint generation, so that solving can proceed unchanged from the original OutsideIn(X) formulation. This phase is called the *prenexer* because it moves the $\exists$-quantifiers leftward, towards the front of the constraint expression — a logical formula is considered to be in *prenex normal form* if all quantifiers are moved to the leftmost, outermost position possible.

Despite the name, the prenexer does not necessarily leave constraints in such a normal form due to the presence of other quantifiers in the formula, specifically the existential quantifiers introduced by local assumption constraints. Observe the rule Prenex$_3$, where a new type variable is bound within an implication constraint. Here, the variable bound by the $\exists\alpha$ quantifier is added to the set of variables bound by the existential quantifier $\exists\bar{\beta}$. This rule therefore ensures that local assumption forms again bind all unification variables introduced within the implication, as they do in the original OutsideIn(X) formulation.

Ordinarily with a explicitly named representation of type variables, we would have to take care when performing these rewrites that no name conflicts occur. As our representation is based on de Bruijn indices, this is quite

[9] We also have to introduce fresh unification variables for all those bound in the user's general type ($\bar{a}$). As they are bound *outside* the implication constraint, they are treated as skolem variables within it. Therefore, this change does not affect the semantics of the original rule.

$$\begin{array}{llll} \text{implication constraints} & I & ::= & \epsilon \mid \exists^I \bar{\alpha}.\ Q \supset \mathcal{C} \mid I \wedge^I I \\ \text{separated constraints} & \mathcal{C} & ::= & Q \cdot I \end{array}$$

$$\begin{array}{lcl} \textbf{implic}[C_1 \wedge' C_2] & = & \textbf{implic}[C_1] \wedge^I \textbf{implic}[C_2] \\ \textbf{implic}[Q] & = & \epsilon \\ \textbf{implic}[\exists \bar{\beta}.\ Q \supset C] & = & \exists^I \bar{\beta}.\ Q \supset \textbf{sep}[C] \\ & & \\ \textbf{simple}[C_1 \wedge' C_2] & = & \textbf{simple}[C_1] \wedge \textbf{simple}[C_2] \\ \textbf{simple}[Q] & = & Q \\ \textbf{simple}[\exists \bar{\beta}.\ Q \supset C] & = & \epsilon \\ & & \\ \textbf{sep}[C] & = & \textbf{simple}[C] \cdot \textbf{implic}[C] \end{array}$$

$$\boxed{\mathcal{Q}; Q_{\text{given}}; \overline{\alpha_{\text{tch}}} \overset{\text{solv}}{\mapsto} \mathcal{C}_{\text{wanted}} \rightsquigarrow Q_{\text{residual}}; \theta}$$

$$\frac{\mathcal{Q}; Q_g; \bar{\alpha} \overset{\text{simp}}{\mapsto} Q \rightsquigarrow Q_r; \theta \qquad \forall ((\exists^I \bar{\beta}_i.\ Q_i \supset \mathcal{C}_i) \in I).\ \mathcal{Q}; Q_g \wedge Q_r \wedge Q_i; \bar{\beta}_i \overset{\text{solv}}{\mapsto} \mathcal{C}_i \rightsquigarrow \epsilon; \theta_i}{\mathcal{Q}; Q_g; \bar{\alpha} \overset{\text{solv}}{\mapsto} Q \cdot I \rightsquigarrow Q_r; \theta}$$

Fig. 5: Solver, separated constraints and separator functions.

mechanical. As each quantifier is moved out, we adjust the indices within each expression accordingly, thus ensuring that names remain unique (See chapter 3 for details).

Once all $\exists$ quantifiers have been moved by the prenexer, we will have an expression of the form $\exists \bar{\alpha}.\ C$ where C consists only of the extended syntax used originally in OUTSIDEIN(X), and all fresh names introduced by the constraint generation have been made globally unique — exactly the semantics of the informal **fresh** constructor used previously. Therefore, after rewriting, we can simply pass C to the subsequent solver phase, using $\bar{\alpha}$ as the set of variables the solver may unify (see the top level rules in Fig. 6).

2.6 Solver and Separator

Our presentation of the solver infrastructure is somewhat different to the solver infrastructure in the original paper. Here we introduce a new form of *separated* C-constraint, denoted $\mathcal{C}$, which contains a vanilla Q-constraint, already solvable by the provided simplifier, and a special I constraint, which contains only local assumption implications (whose bodies are, in turn, separated $\mathcal{C}$ constraints, see Fig. 5). Our solver operates on these separated constraints rather than the C-constraints directly, so we introduce a function **sep** : $C \to \mathcal{C}$, defined in terms of helper functions **simple** : $C \to Q$ and **implic** : $C \to I$, which forms an additional phase in the inference algorithm.

In the original presentation of the OUTSIDEIN(X) solver, the separation of constraints into simple and implication components was performed inline with the main solver rule, rather than in a separate phase. This makes the termination argument for the main solving rule slightly less clear, as it must be established that, for all C, **simple**[C] and **implic**[C] are no larger than C itself. This, while obvious to a human observer, is not so obvious to a proof assistant. We simply sidestep any termination trouble by encoding the separation of constraints as a separate phase, rather than interleaving them as in the original presentation.

Our main solving judgement is of the following form:

$$\mathcal{Q}; Q_{\text{given}}; \overline{\alpha_{\text{tch}}} \overset{\text{solv}}{\mapsto} \mathcal{C}_{\text{wanted}} \rightsquigarrow Q_{\text{residual}}; \theta$$

This can be read as, "Given the axiom scheme $\mathcal{Q}$ and constraint Q_{given}, the constraint $\mathcal{C}_{\text{wanted}}$ is simplified to Q_{residual} producing the substitution θ where $dom(\theta) \subseteq \overline{\alpha_{\text{tch}}}$." Note that while the solver is not obliged to solve *all* constraints, the remaining residual constraint is a Q-constraint, not a $\mathcal{C}$-constraint, which means that, at the very least, all implication constraints must be resolved. This is important, as the constraints left residual from the solver are abstracted over when generalising on top-level definitions (see Fig. 6), and C constraints do not form part of the constraint language available to the user of the language — It would be highly unusual for a type inference algorithm to provide type signatures that the user of the language could not express themselves!

$$\boxed{\mathcal{Q}; \Delta; \Gamma \mapsto \mathit{prog}}$$

$$\frac{}{\mathcal{Q}; \Delta; \Gamma \mapsto \epsilon}\ \textsc{Empty} \qquad \frac{\begin{array}{c}\bar{a}, \Delta; \Gamma \mapsto e : \tau \rightsquigarrow C \qquad C \overset{\text{pnx}}{\hookrightarrow} \exists \bar{\beta}.\ C' \qquad \mathcal{Q}; Q; \bar{\beta} \overset{\text{solv}}{\mapsto} \mathbf{sep}[C'] \rightsquigarrow \epsilon; \theta \\ \mathcal{Q}; \Delta; \Gamma, (f : \forall \bar{a}.\ Q \Rightarrow \tau) \mapsto \mathit{prog}\end{array}}{\mathcal{Q}; \Delta; \Gamma \mapsto f :: (\forall \bar{a}.\ Q \Rightarrow \tau) = e, \mathit{prog}}\ \textsc{BindA}$$

$$\frac{\begin{array}{c}\gamma, \Delta; \Gamma \mapsto e : \gamma \rightsquigarrow C \qquad C \overset{\text{pnx}}{\hookrightarrow} \exists \bar{\beta}.\ C' \qquad \mathcal{Q}; Q; \gamma, \bar{\beta} \overset{\text{solv}}{\mapsto} \mathbf{sep}[C'] \rightsquigarrow Q_r; \theta \\ \bar{\alpha} = \text{free unification variables in } Q_r, \theta\gamma \qquad \mathcal{Q}; \Delta; \Gamma, (f : \forall \bar{\alpha}.\ Q_r \Rightarrow \theta\gamma) \mapsto \mathit{prog}\end{array}}{\mathcal{Q}; \Delta; \Gamma \mapsto f = e, \mathit{prog}}\ \textsc{Bind}$$

Fig. 6: Top level algorithmic rules.

2.7 Top Level Rules

Our presentation of the top-level rules differ significantly from the original OutsideIn(X) presentation, in order to accommodate the other changes we have made to the system. In particular:

- As we view the type in the constraint generation rule as *input* rather than *output*, we do not need to add an equality constraint in the rule BindA, reconciling the provided type signature with the generated type. Instead, we simply pass the provided type directly in to constraint generation. Similarly, we cannot simply use the type returned by constraint generation in the rule Bind, but must instead introduce a new "fresh" type variable γ, use it for constraint generation, then apply the solution substitution θ to it in order to determine the type of f.
- As we now have an explicit notion of available type variables in the environment Δ for constraint generation, we add a similar environment here.
- Constraints must be prenexed *and* separated before being solved, which necessitates some additions to the Bind rules.

Despite the added rigour in our version of the OutsideIn(X) system, some informality still remains in these definitions - in particular, the possibility of name conflicts is ignored, assertions are made informally about substitution domains, and the rule Bind in Figure 6 relies on an informally specified "free unification variables" operation. Naturally, in order to formalise this system in a proof assistant, we still must resolve these issues. All of these problematic elements are eliminated via our representation of terms and names, discussed in the next chapter.

3 Encoding Types and Terms

Constraints and expressions cannot be expressed in Agda simply as a series of data types, because the exact structure of these terms is dependent on the parameter X. C-constraints, which are not part of X, can contain Q-constraints, which are part of it. Similarly, expressions, not part of X, can contain types, which are.

We use Agda's module system to represent this parameterisation, providing a record type X containing all definitions within the parameter as an argument to our modules. Here is a sketch of the Agda code used to represent such a parameterisation:

```
record X : Set₁ where
    field Type : ···
    field QConstraint : ···
    ...

module OutsideIn (x : X) where
    open X(x)
    ...
    data CONSTRAINT ···
    data EXPRESSION ···
```

(In all future code, we will use sans-serif font to refer to elements of the parameter X)

Note that the type of X is Set_1, rather than the usual type-of-types Set, because the X parameter sits one meta-level higher than the definitions themselves — the type Set cannot contain Set without making Agda inconsistent[10].

Over the course of this chapter, we will gradually refine this sketch into the concrete definitions we use in our formalisation.

3.1 Names

One of the most commonly examined facets of term representation is how to represent variable names. Much literature has been published on the subject, and a wide range of techniques exist. Perhaps the most common is that of the *de Bruijn index* (de Bruijn, 1972), a simple system of assigning numerical indices to binders instead of names. For example, the term $\lambda x.\ \lambda y.\ x\ y$ can be restated with (stack-based) de Bruijn indices as $\lambda.\ \lambda.$ `1 0`. These indices are sometimes presented the other way around, where the innermost binder is referenced by the *highest* available index, but for our purposes, this orientation is easier.

These indices make reasoning and manipulating terms substantially easier in many cases: avoiding name clashes when rewriting constraints is simply a matter of small arithmetic operations on indices, and α-equivalent terms are propositionally equal.

Nicolas Pouillard has generalised de Bruijn indices in a series of systems (implemented in Agda, no less), starting with "Nameless, Painless", published in (Pouillard, 2011). Based on the notion of an *abstract world* of variable names, these systems are designed chiefly to avoid programming errors when working with de Bruijn indices (which, over the years, have established some notoriety for being somewhat difficult beasts to tame). While his approach is certainly not without merit, we feel that using such a library to represent terms in OUTSIDEIN(X) may needlessly complicate our definitions. The approach we have taken, based on de Bruijn indices, allows us to exploit type-indexing techniques to generate a number of "theorems for free" via parametricity (Wadler, 1989), and expose an elegant categorical structure in our representation. It is not clear that we could retain this simplicity and generality were we to rely on Pouillard's work.

[10] The inconsistency arises from Russell's paradox.

3.2 Indexed Terms

The simplest possible representation which uses de Bruijn indices simply uses the full set ℕ to represent type variables:[11]

```
record X : Set₁ where
    field  Type          :  Set
           Var           :  ℕ → Type
           →′            :  Type → Type → Type
    field  QConstraint   :  Set
           ϵ             :  QConstraint
           ∧             :  QConstraint → QConstraint → QConstraint
           ∼             :  Type → Type → QConstraint

module OutsideIn (x : X) where
    open X(x)
    data CONSTRAINT : Set where
        QC          :  QConstraint → CONSTRAINT
        ∧′          :  CONSTRAINT → CONSTRAINT → CONSTRAINT
        Ǝ           :  CONSTRAINT → CONSTRAINT
        ∃_._ ⊃ _    :  ℕ → QConstraint → CONSTRAINT → CONSTRAINT
```

Fig. 7: Naïve constraint representation

This encoding has a number of obvious problems. For example, all terms have an infinite number of free variables available, as the full type ℕ is used for variable names. This makes it impossible to determine instantly whether a term is closed or if a term contains free variables; one must instead analyse the term to extract this information. A common technique used to solve this problem when encoding binders in dependently typed languages is to index the type of terms by the number of available variables in the term. This technique is used often in generic programming literature, such as (Morris *et al.*, 2004), and was also shown by McBride to provide a convenient termination measure that can be used to phrase first-order unification as structural recursion (McBride, 2003). Reworking the above term definition to include such indexing, we get:

```
record X : Set₁ where
    field  Type          :  ℕ → Set
           Var           :  ∀{n : ℕ} → FIN n → Type n
           →′            :  ∀{n : ℕ} → Type n → Type n → Type n
    field  QConstraint   :  ℕ → Set
           ϵ             :  ∀{n : ℕ} → QConstraint n
           ∧             :  ∀{n : ℕ} → QConstraint n → QConstraint n → QConstraint n
           ∼             :  ∀{n : ℕ} → Type n → Type n → QConstraint n

module OutsideIn (x : X) where
    open X(x)
    data CONSTRAINT (n : ℕ) : Set where
        QC          :  QConstraint n → CONSTRAINT n
        ∧′          :  CONSTRAINT n → CONSTRAINT n → CONSTRAINT n
        Ǝ           :  CONSTRAINT (suc n) → CONSTRAINT n
        ∃_._ ⊃ _    :  (m : ℕ) → QConstraint n → CONSTRAINT (n + m) → CONSTRAINT n
```

Fig. 8: FIN-named constraint representation

[11] Where ℕ is the type of the standard Peano naturals with zero : ℕ and suc : ℕ → ℕ

This definition provides a type-level distinction between closed terms and terms that may contain some free variables, eliminating the problems with the earlier encoding. It enforces this by demanding that type variables be of type $\textsc{Fin}\ n$, where n is the number of available variables in the term. $\textsc{Fin}\ n$ is a type containing exactly n inhabitants — a finite set of natural numbers $[0, n)$ — defined as follows:

$$\begin{array}{lll}\textbf{data} & \multicolumn{2}{l}{\textsc{Fin} : \mathbb{N} \to \text{Set } \textbf{where}} \\ & \textit{zero} & : \ \forall\{n : \mathbb{N}\} \to \textsc{Fin}\ (\text{suc}\ n) \\ & \textit{suc} & : \ \forall\{n : \mathbb{N}\} \to \textsc{Fin}\ n \to \textsc{Fin}\ (\text{suc}\ n)\end{array}$$

With this definition, the previously infinite number of available variables is now restricted to a finite number described by the type index. For example, the term $\exists.\ \exists.\ QC\ (\mathsf{Var}\ \textit{zero} \sim \mathsf{Var}\ (\textit{suc zero}))$ is closed and could therefore be of type $\textsc{Constraint}\ n$ for any $n : \mathbb{N}$ — that is, it could appear in a context with any number of available variables (including zero). The body of that constraint, $QC\ (\mathsf{Var}\ \textit{zero} \sim \mathsf{Var}\ (\textit{suc zero}))$, is by contrast typed most generally as $\textsc{Constraint}\ (\text{suc}\ (\text{suc}\ n))$ for any $n : \mathbb{N}$, which means that the term can only validly appear in a context with two or more available variables. Therefore, our form for the $\exists$ quantifier *introduces* a new type variable by incrementing the index:

$$\exists : \textsc{Constraint}\ (\text{suc}\ n) \to \textsc{Constraint}\ n$$

The local assumption constraint, unlike the $\exists$ quantifier, introduces more than one variable at a time:

$$\exists\_.\_ \supset \_ : (m : \mathbb{N}) \to \mathsf{QConstraint}\ n \to \textsc{Constraint}\ (n + m) \to \textsc{Constraint}\ n$$

Note that the antecedent QConstraint has only n available variables, not $n + m$, as it is impossible for the antecent to mention any unification variables introduced in the succedent C-constraint.

3.3 Nested Data Types

While this indexing provides us a very nice way to handle de Bruijn indices for bound variables, we have not introduced an elegant way to handle type constructors, such as *Int* or *Maybe*, which exist in the top-level environment.

From the perspective of the constraint generation and solver infrastructure, type constructors are no different from type variables. One possible solution, therefore, is to simply assign indices to these top level type constructors. This makes type constructors difficult to distinguish from type variables, however, which becomes a serious problem when instantiating the X parameter — type constructors, which are treated as *rigid* and cannot be unified, must be distinguished from unification type variables, which can be substituted. In addition, assigning indices to top-level global definitions is aesthetically unpleasing.

A common alternative is to introduce a separate *Con* introduction form for Type, which refers to type constructors, as opposed to the *Var* form for type variables. This solution becomes unsatisfactory when we examine the solver infrastructure (see Fig. 5). Note that, when solving each implication constraint, *all* variables bound outside the implication constraint, *including* variables that are unifiable outside the implication, are treated as rigid, skolem variables. This means that a variable previously treated as unifiable could, in a different context, be treated as rigid like a constructor. Using a separate *Con* form therefore does not bring any advantage — we still have some subset of the available type variables being treated as skolem.

Our approach allows us to treat type constructors and type variables identically in the constraint generation and solver infrastructure of the $\textsc{OutsideIn}(X)$ system itself, but retain the ability to separate skolem variables from unification variables when implementing the simplifier. In addition, type constructors can be represented by *any* type[12], and therefore do not need to be assigned indices, which makes working with the system a great deal more convenient.

[12] Provided such a type has decidable equality.

Our approach is to index terms not by the *size* of the set of available type variables, but by the *set itself*, as shown below:

$$
\begin{array}{l}
\textbf{record } \mathrm{X} : \mathrm{Set}_1 \textbf{ where} \\
\quad \begin{array}{lll}
\textbf{field} & \mathsf{Type} & : \ \mathrm{Set} \to \mathrm{Set} \\
 & \mathsf{Var} & : \ \forall\{n : \mathrm{Set}\} \to n \to \mathsf{Type}\ n \\
 & \to' & : \ \forall\{n : \mathrm{Set}\} \to \mathsf{Type}\ n \to \mathsf{Type}\ n \to \mathsf{Type}\ n \\
\textbf{field} & \mathsf{QConstraint} & : \ \mathrm{Set} \to \mathrm{Set} \\
 & \epsilon & : \ \forall\{n : \mathrm{Set}\} \to \mathsf{QConstraint}\ n \\
 & \wedge & : \ \forall\{n : \mathrm{Set}\} \to \mathsf{QConstraint}\ n \to \mathsf{QConstraint}\ n \to \mathsf{QConstraint}\ n \\
 & \sim & : \ \forall\{n : \mathrm{Set}\} \to \mathsf{Type}\ n \to \mathsf{Type}\ n \to \mathsf{QConstraint}\ n
\end{array} \\
\\
\textbf{module } \mathit{OutsideIn}\ (x : \mathrm{X}) \textbf{ where} \\
\quad \textbf{open } X(x) \\
\quad \textbf{data } \textsc{Constraint}\ (n : \mathrm{Set}) : \mathrm{Set} \textbf{ where} \\
\quad\quad \begin{array}{ll}
QC & : \ \mathsf{QConstraint}\ n \to \textsc{Constraint}\ n \\
\wedge' & : \ \textsc{Constraint}\ n \to \textsc{Constraint}\ n \to \textsc{Constraint}\ n \\
\exists & : \ \textsc{Constraint}\ (\mathcal{S}\ n) \to \textsc{Constraint}\ n \\
\exists\_.\_ \supset \_ & : \ (m : \mathbb{N}) \to \mathsf{QConstraint}\ n \to \textsc{Constraint}\ (n \oplus m) \to \textsc{Constraint}\ n
\end{array}
\end{array}
$$

Fig. 9: Constraint representation with nested data types

The secret to this representation lies in the $\mathcal{S}$ data type, used when new type variables are made available by quantifiers. As an additional bound variable is now available, the type $\mathcal{S}\ \tau$ must be isomorphic to $\tau + 1$[13], and is therefore implemented as follows:

$$
\begin{array}{lll}
\textbf{data} & \mathcal{S}\ (\tau : \mathrm{Set}) : \mathrm{Set} \textbf{ where} & \\
 & \quad \mathit{zero} \ : \ \mathcal{S}\ \tau & \\
 & \quad \mathit{suc} \ \ : \ \tau \to \mathcal{S}\ \tau &
\end{array}
$$

We also introduce another operation $\oplus$, for local assumptions, which is essentially repeated application of $\mathcal{S}$:

$$
\begin{array}{lcl}
\_\oplus\_ & : & \mathit{Set} \to \mathbb{N} \to \mathit{Set} \\
x \oplus \mathrm{zero} & = & x \\
x \oplus \mathrm{suc}\ n & = & (\mathcal{S}\ x) \oplus n
\end{array}
$$

Then, a simplifier can take as arguments constraints of type $\mathsf{QConstraint}\ (x \oplus n)$, where x is the type for rigid variables and constructors, and n is the number of additional variables that can be unified. This obviates the need for an explicit set $\overline{\alpha_{\mathrm{tch}}}$ to be passed to the solver — instead this information can be gleaned purely from the type of constraints passed in. This approach is advantageous because it allows us to handle this reinterpretation of variables in a unification or skolem context without changing the structure of the term.

3.3.1 Monads

Another elegant observation about this representation is that type terms can now form a categorical abstraction familiar to every Haskell programmer — a *monad*. This monadic structure is not an original discovery; it has been demonstrated by others in the past, doing similar work on term representation (Bird & Paterson, 1999; Bellegard & Hook, 1994). Originally from category theory, a *monad* is understood by functional programmers to be a type constructor m, a function $\mathit{unit} : \alpha \to m\ \alpha$ and a *Kleisli composition* operator $\circ_m : (\beta \to m\ \gamma) \to (\alpha \to m\ \beta) \to (\alpha \to m\ \gamma)$, such that the following laws hold:

$$
\begin{array}{lrcl}
\textbf{Left Identity}: & \mathit{unit} \circ_m f & \doteq & f \\
\textbf{Right Identity}: & g \circ_m \mathit{unit} & \doteq & g \\
\textbf{Associativity}: & f \circ_m (g \circ_m h) & \doteq & (f \circ_m g) \circ_m h
\end{array}
$$

[13] i.e. Haskell's `Maybe` type.

A *category* is comprised of a class of *objects*, a class of *arrows* or *morphisms* between those objects, including an identity morphism for each object, and a morphism composition operation $\circ$ which must be an associative and respect identity morphisms. From this definition, it is obvious that the above *monad laws* are just restatements of the *category laws* for a specific category, called the *Kleisli category* for the monad m.

We introduce informally the notion of a category **Agda**, consisting of Agda types as objects, Agda functions as morphisms, $(\lambda x \to x)$ as every identity morphism and function composition $\circ$ as morphism composition.[14] Then, the Kleisli category for Type is a *subcategory* of **Agda** with all the same objects, but only those functions with types of the form $\alpha \to \mathsf{Type}\ \beta$ as morphisms. Note that the type $\alpha \to \mathsf{Type}\ \beta$ is that of a substitution; Kleisli composition is substitution composition, where Var is the identity substitution. Var is therefore the *unit* operation for our monad and the identity morphism for our Kleisli category.

By requiring in our X parameter that Type be a monad, we are able to assume that substitution is well-behaved with respect to the structure of the type terms.

3.3.2 Functors

We can simply derive the familiar *bind* function from Kleisli composition and *unit*:

$$\begin{aligned}&\mathit{bind} : \forall\{\alpha\ \beta\} \to (\alpha \to m\ \beta) \to m\ \alpha \to m\ \beta\\ &\mathit{bind}\ f\ a = (f \circ_m (\lambda x \to a))\ \text{it}\end{aligned}$$

(Where "it" is a constructor for a unit type)

When applied to types, *bind* is clearly application of a substitution to a term:

$$\mathit{bind} : \forall\{\alpha\ \beta\} \to (\alpha \to \mathsf{Type}\ \beta) \to \mathsf{Type}\ \alpha \to \mathsf{Type}\ \beta$$

Viewed categorically, *bind* could be viewed here as a mapping from morphisms in the Kleisli category of Type (i.e functions with types of the form $\alpha \to \mathsf{Type}\ \beta$) to morphisms in a new subcategory of **Agda** in the image of Type, which we call the Type-subcategory. This category has morphisms consisting of Agda functions, and objects consisting of types of the form $\mathsf{Type}\ \tau$ for any τ. Note that the following two properties hold for *bind*:

$$\begin{array}{lll}\textbf{Identity}: & \mathit{bind\ unit} & \doteq\ \ \mathit{id}\\ \textbf{Composition}: & \mathit{bind}\ f \circ \mathit{bind}\ g & \doteq\ \ \mathit{bind}\ (f \circ_{\mathsf{Type}} g)\end{array}$$

If we have a mapping f from a category A to a category B, such that identity maps to identity, and composition maps to composition, then f is called a *functor* from A to B. As this is true of *bind*, we can say that *bind* is a functor from the Kleisli category of Type to the Type-subcategory.

We would like to be able to perform substitution on more than just Type terms. Specifically, it is also necessary to perform substitution on QConstraints and, ultimately, CONSTRAINTS.

To achieve this we require in the parameter X *another* functor, also from the Kleisli category of Type, but this time to the subcategory of **Agda** in the image of QConstraint — the QConstraint-subcategory:

$$\mathit{Q\text{-}subst} : \forall\{\alpha\ \beta\} \to (\alpha \to \mathsf{Type}\ \beta) \to (\mathsf{QConstraint}\ \alpha \to \mathsf{QConstraint}\ \beta)$$

The functor laws tell us that this, once again, respects substitution operations:

$$\begin{array}{lll}\textbf{Identity}: & \mathit{Q\text{-}subst\ unit} & \doteq\ \ \mathit{id}\\ \textbf{Composition}: & \mathit{Q\text{-}subst}\ f \circ \mathit{Q\text{-}subst}\ g & \doteq\ \ \mathit{Q\text{-}subst}\ (f \circ_{\mathsf{Type}} g)\end{array}$$

Given this functor, similar functors can be produced for CONSTRAINT and, indeed, any type that similarly contains QCONSTRAINTS.

Some difficulty arises, however, when defining the functor for terms which introduce variables: The form $\exists.\ C$ contains a subexpression of type CONSTRAINT $(\mathcal{S}\ \alpha)$. Showing that $\mathcal{S}$ is a monad[15] produces the requisite

[14] This is similar to the imaginary category **Hask** for Haskell, however unlike **Hask**, Agda types and functions *do* actually form a category

[15] The familiar `Maybe` monad, no less

functors to transform our substitution $\alpha \to \mathsf{Type}\ \beta$ to a substitution $\mathcal{S}\ \alpha \to \mathsf{Type}\ (\mathcal{S}\ \beta)$, as required for this case. The form $\exists n.\ Q \supset C$ contains a subexpression of type $\textsc{Constraint}\ (\alpha\ \oplus\ n)$, which is a slightly more complicated beast to tame. Here, we must show not only that $\mathcal{S}$ is a monad, but that $\mathcal{S} \circ \mathcal{S}$ is also, and $\mathcal{S} \circ \mathcal{S} \circ \mathcal{S}$, and so on. In other words, we have to show that for all n, $\lambda\alpha.\ \alpha \oplus n$ is a monad. Towards this end, we borrow the concept of a *monad transformer*, i.e. a monad homomorphism *lift* from any monad M to $M \circ \mathcal{S}$. By showing that *lift* is monad homomorphism for the $\mathcal{S}$-transformer, and that the resultant type $M \circ \mathcal{S}$ is a monad if M is a monad, it can be trivially shown that any number of $\mathcal{S}$'s form a monad, and thus that $\lambda\alpha.\ \alpha \oplus n$ is a monad as required.

Using the above, we have defined similar "substitution functors" for all types that are indexed by the set of type variables.

3.3.3 Renaming

Another common operation for Types, $\mathsf{QConstraint}$s and so on is *renaming.* Renaming can be implemented via functor composition in terms of substitution:

$$\begin{aligned}
&\tau\textit{-rename} : \forall\{\alpha\ \beta\} \to (\alpha \to \beta) \to (\mathsf{Type}\ \alpha \to \mathsf{Type}\ \beta)\\
&\tau\textit{-rename}\ f\ =\ \textit{bind}\ (\textit{rename}\ f)\\
&\\
&Q\textit{-rename} : \forall\{\alpha\ \beta\} \to (\alpha \to \beta) \to (\mathsf{QConstraint}\ \alpha \to \mathsf{QConstraint}\ \beta)\\
&Q\textit{-rename}\ f\ =\ Q\textit{-subst}\ (\textit{rename}\ f)\\
&\\
&\dots
\end{aligned}$$

Where *rename* is a functor from the base **Agda** category to the Kleisli category of Type:

$$\begin{aligned}
&\textit{rename} : \forall\{\alpha\ \beta\} \to (\alpha \to \beta) \to (\alpha \to \mathsf{Type}\ \beta)\\
&\textit{rename}\ f\ =\ \textit{unit} \circ f
\end{aligned}$$

With these definitions, the common de Bruijn index "up-shift" substitution which increments every index is trivially *rename suc*. Up-shifting by multiple variables at a time can be done simply by *rename* $\textit{unit}_n$ where $\textit{unit}_n$ is the *unit* morphism of the monad $\lambda\alpha.\ \alpha \oplus n$.

3.3.4 A Note on Equality

Propositional equality is defined simply in Agda[16], as a data type that reifies definitional equality:

$$\begin{aligned}
&\textbf{data}\ \ \equiv \{A : \mathrm{Set}\} : A \to A \to \mathrm{Set}\ \textbf{where}\\
&\qquad \textit{refl} : \forall\{x : A\} \to x \equiv x
\end{aligned}$$

This allows us to introduce definitional equality constraints to the local context by pattern matching, in order to prove basic theorems like transitivity and symmetry of equality:

$$\begin{aligned}
&\textit{trans} : \forall\{A : \mathrm{Set}\}\{x\ y\ z : A\} \to x \equiv y \to y \equiv z \to x \equiv z\\
&\textit{trans}\ \textit{refl}\ \textit{refl} = \textit{refl}\\
&\\
&\textit{sym} : \forall\{A : \mathrm{Set}\}\{x\ y : A\} \to x \equiv y \to y \equiv z\\
&\textit{sym}\ \textit{refl} = \textit{refl}
\end{aligned}$$

The monad and functor laws shown above, however, are in terms of extensional equality, not this propositional equality. Unfortunately, this equality is not extensional — that is, the statement that two functions f and g are

[16] This definition is somewhat simplified - in particular, universe polymorphism is removed.

propositionally equal, if, for all x, $f(x)$ is propositionally equal to $g(x)$ is not provable in Agda. Any value x can only be said to be propositionally equal to some other value y if x and y both normalise to the same result. If the definitions of two functions are (intensionally) different, they will not normalise to the same definition, even if they give the same result for all inputs.

The general approach for proving lemmas which require extensionality is to prove them within Altenkirch's setoid-based[17] model (Altenkirch, 1999). As this becomes quite tedious in practice, we opt for the perhaps more inelegant approach of simply postulating functional extensionality:

$$\textbf{postulate}\ \textit{extensionality} : \{A\ B : \text{Set}\}\{f\ g : A \rightarrow B\} \rightarrow (\forall x \rightarrow f\ x \equiv g\ x) \rightarrow f \equiv g$$

Agda's logic is consistent with this postulate, however we do lose the sometimes-valuable property of *canonicity* for equality proofs – that is, not all equality proofs normalise to a canonical closed term (i.e *refl*), as the postulate prevents normalisation. In practice, this means that programs which depend on this postulate will not give meaningful results (they crash the program much like Haskell's `error`), however the postulate can be freely used for proof work - that is, to show that a particular type is inhabited.

We do not need extensionality to model the OutsideIn(X) algorithm itself. We require it only to prove monad laws and similar identities about our term representation. The loss of canonicity in exchange for simpler proofs is a trade we deem acceptable.

3.4 Separator and Prenexer

The definition of SeparatedConstraint, our representation of $\mathcal{C}$, is uninteresting save that it too is indexed by the set of available type variables, and is similarly equipped with functors for substitution.

It seems correct, then, that our separator should be a total function of the following type:

$$\textit{sep} : \forall\ \{\textit{tv} : \text{Set}\} \rightarrow \textsc{Constraint}\ \textit{tv} \rightarrow \textsc{SeparatedConstraint}\ \textit{tv}$$

However our *sep* function is not total, as it is. The **sep** function defined in Figure 5 is not defined for constraints of the form Ⅎ. C. To solve this problem, we adjust our constraint representation to be indexed by a *strata*:

```
record X : Set₁ where
    field  Type         :  Set → Set
           Var          :  ∀{n : Set} → n → Type n
           →′           :  ∀{n : Set} → Type n → Type n → Type n
    field  QConstraint  :  Set → Set
           ϵ            :  ∀{n : Set} → QConstraint n
           ∧            :  ∀{n : Set} → QConstraint n → QConstraint n → QConstraint n
           ∼            :  ∀{n : Set} → Type n → Type n → QConstraint n
module OutsideIn (x : X) where
    open X(x)
      data Strata : Set where
          Hi  :  Strata
          Lo  :  Strata
    data Constraint (n : Set) : Strata → Set where
        QC           :  ∀{σ : Strata} → QConstraint n → Constraint n σ
        ∧′           :  ∀{σ : Strata} → Constraint n σ → Constraint n σ → Constraint n σ
        Ⅎ            :  Constraint (𝒮 n) Hi → Constraint n Hi
        ∃_._ ⊃ _     :  ∀{σ} → (m : ℕ) → QConstraint n → Constraint (n ⊕ m) σ
                        → Constraint n σ
```

Fig. 10: Final Constraint Representation

[17] A *setoid* being a dependent product $(\tau, \approx)$ where $\approx$ is an equivalence relation on the type τ.

$$\boxed{C \xrightarrow{\text{prenex}} C}$$

$$\frac{}{Q \xrightarrow{\text{prenex}} Q}\ \textsc{PrenexQC} \qquad \frac{C \xrightarrow{\text{prenex}} C'}{\exists\alpha.\ C \xrightarrow{\text{prenex}} \exists\alpha.\ C'}\ \textsc{PrenexF}$$

$$\frac{x \xrightarrow{\text{prenex}} \exists\bar{\alpha}.\ x' \qquad y \xrightarrow{\text{prenex}} \exists\bar{\beta}.\ y'}{x \wedge y \xrightarrow{\text{prenex}} \exists\bar{\alpha}\bar{\beta}.\ x' \wedge y'}\ \textsc{PrenexConj}$$

$$\frac{C \xrightarrow{\text{prenex}} \exists\bar{\beta}.\ C'}{\exists\bar{\alpha}.\ Q \supset C \xrightarrow{\text{prenex}} \exists\bar{\alpha}\bar{\beta}.\ Q \supset C'}\ \textsc{PrenexImp}$$

Fig. 11: A structurally recursive prenexer

Note that the type Constraint *n Hi* contains all forms of constraints, whereas the type Constraint *n Lo* contains all constraints except for constraints of the form $\exists.\ C$. This allows us to achieve a limited kind of subtyping via type indexing.

Now we can more accurately specify the domain of the *sep* function, so as to make it total:

$$sep : \forall\ \{tv : \text{Set}\} \to \textsc{Constraint}\ tv\ Lo \to \textsc{SeparatedConstraint}\ tv$$

We can also give a more detailed account of the type of the prenexer:

$$prenex : \forall\ \{tv : \text{Set}\} \to \textsc{Constraint}\ tv\ Hi \to \exists\ (\lambda\ n \to \textsc{Constraint}\ (tv \oplus n)\ Lo)$$

Implementing the prenexer as presented in Figure 4 is somewhat difficult, as it is presented as a rewrite system rather than as a neat structural recursion. We instead implement a structurally recursive prenexer as shown in Figure 11. The equivalence between these two formalisations can be proven via an easy induction.

3.5 Expressions

Our representation of Expression merits some attention as it is indexed by not one, but *two* types — one for available type variable names, and one for available variables on the *value* level:

$$\textbf{data}\ \textsc{Expression}\ (ev\ tv : \text{Set}) :\ \cdots$$

This approach brings a number of advantages. For example, type environments like Γ can be represented as a total function — it is impossible for an environment lookup to fail. This also ensures that we update the type environment whenever new value-level variables are available.

It is problematic, however, when dealing with pattern matching. Inside a pattern alternative, some number of new value variables are introduced into scope — this number should be reflected in the *ev* index. The exact number of variables, however, is dependent on the *type* (or, at least, the arity) of the data constructor in the pattern, which is not information we have in our possession when constructing syntax trees. Furthermore, it is already clear that data constructors require separate treatment from regular variables, as patterns contain only names of data constructors, not arbitrary variables.

This calls for a more sophisticated representation of names in expressions. Instead of simply using *ev* to refer to value-level variables, we will use Name *ev*, defined as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{data}\ \textsc{NameType} : \text{Set}\ \textbf{where}} \\ \quad Binding & : & \textsc{NameType} \\ \quad Datacon & : & \mathbb{N} \to \textsc{NameType} \end{array}$$

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{data}\ \textsc{Name}\ (n : \text{Set})\ :\ \textsc{NameType} \to \text{Set}\ \textbf{where}} \\ \quad N & : & n \to \textsc{Name}\ n\ Binding \\ \quad DC & : & \forall\ \{x\} \to \mathsf{dc}\ x \to \textsc{Name}\ n\ (Datacon\ x) \end{array}$$

(Where dc is a type for datacon names, indexed by their arity, provided in the parameter X)

A Name can signify, therefore, either a straightforward binding *or* a data constructor of some arity. As the arity is presented in the type Name, we can easily produce the desired type for a pattern matching alternative:

$$\begin{array}{l} \textbf{data}\ \textsc{Alternative}\ (ev\ tv : \text{Set}) : \textsc{Shape} \to \text{Set}\ \textbf{where} \\ \quad \xrightarrow{\text{alt}}\ :\ \forall\{n : \mathbb{N}\} \to \textsc{Name}\ ev\ (\mathit{Datacon}\ n) \to \textsc{Expression}\ (ev\ \oplus\ n)\ tv\ \to \\ \qquad\quad \textsc{Alternative}\ ev\ tv \end{array}$$

This technique works greatly to our advantage as there is another instance where data constructors require special treatment; this time in constraint generation. Observe the two rules in constraint generation where the environment is consulted; the rules VarCon and Case:

$$\frac{(v : \forall \bar{a}.\ Q_1 \Rightarrow \tau_1)\ \in \Gamma}{\Delta; \Gamma \mapsto v : \tau \rightsquigarrow \exists \bar{\alpha}.\ [\overline{a \mapsto \alpha}]Q_1 \wedge' (\tau \sim [\overline{a \mapsto \alpha}]\tau_1)}\ \textsc{VarCon}$$

$$\frac{\begin{array}{c} \alpha, \beta, \bar{\gamma}, \bar{\delta}, \Delta; \Gamma \mapsto e : \alpha \rightsquigarrow C \\ (K_i : \forall \bar{a}\bar{b}.\ Q_i \Rightarrow \bar{\tau}_i \to \mathtt{T}\ \bar{a})\ \in \Gamma \qquad \alpha, \beta, \bar{\gamma}, \bar{\delta}, \bar{\rho}, \Delta; \Gamma, (\overline{x_i : [b \mapsto \rho][a \mapsto \gamma]\tau_i}) \mapsto e_i : \delta_i \rightsquigarrow C_i \\ C'_i = \begin{cases} C_i \wedge' \delta_i \sim \beta & \text{if } \bar{b}_i = \epsilon \text{ and } Q_i = \epsilon \\ \exists \bar{\rho}.\ \exists \epsilon.([\overline{b \mapsto \rho}][\overline{a \mapsto \gamma}]Q_i) \supset C_i \wedge' \delta_i \sim \beta & \text{otherwise} \end{cases} \end{array}}{\Delta; \Gamma \mapsto \mathtt{case}\ e\ \mathtt{of}\ \{\overline{K_i\ \bar{x}_i \to e_i}\} : \tau \rightsquigarrow \exists \alpha. \exists \beta. \exists \bar{\gamma}. \exists \bar{\delta}.\ C \wedge' (\mathtt{T}\ \bar{\gamma} \sim \alpha) \wedge' (\bigwedge C'_i) \wedge' (\tau \sim \beta)}\ \textsc{Case}$$

Note that in each instance, the form of the type schema retrieved from the environment is different! One form for data constructors ($\forall \bar{a}\bar{b}.\ Q \Rightarrow \bar{\tau} \to \mathtt{T}\ \bar{a}$), and another for bindings ($\forall \bar{a}.\ Q \Rightarrow \tau$). As Type is part of the X parameter, we do not have the luxury of simply pattern-matching on the structure of the type schema to determine if it is of the correct form. Instead we adjust our encoding of type schema, in order to force the user to provide correctly-structured types in the environment for data constructors. Specifically, we index our representation of type schema by a NameType, providing a general type schema form for types of regular *Binding*s but more structured forms for types of data constructors:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{data}\ \textsc{TypeSchema}\ (tv : \text{Set}) : \textsc{NameType} \to \text{Set}\ \textbf{where}} \\ \quad \forall'\_.\_\Rightarrow\_ & : & (n : \mathbb{N}) \to \mathsf{QConstraint}\ (tv\ \oplus\ n) \to \mathsf{Type}\ (tv \oplus n) \to \textsc{TypeSchema}\ tv\ \mathit{Regular} \\ \quad \forall'\_.\_\longrightarrow\_ & : & (a : \mathbb{N})\ \{r : \mathbb{N}\} \to \textsc{Vec}\ (\mathsf{Type}\ (tv\ \oplus\ a))\ r \to tv \to \textsc{TypeSchema}\ tv\ (\mathit{Datacon}\ r) \\ \quad \forall'\_,\_.\_\Rightarrow\_\longrightarrow\_ & : & (a\ b : \mathbb{N})\ \{r : \mathbb{N}\} \to \mathsf{QConstraint}\ (tv\ \oplus\ a\ \oplus\ b) \to \\ & & \textsc{Vec}\ (\mathsf{Type}\ (tv\ \oplus\ a\ \oplus\ b))\ r \to tv \to \textsc{TypeSchema}\ tv\ (\mathit{Datacon}\ r) \end{array}$$

Here, we use a length-indexed vector Vec to ensure that the arity of the function type for data constructors matches the arity mentioned in their Name.

This allows us to define environments as a total mapping of any Name of type n to TypeSchema for the same name type n.

$$\begin{array}{l} \mathit{Environment} : \text{Set} \to \text{Set} \to \text{Set} \\ \mathit{Environment}\ ev\ tv = \forall\{n : \textsc{NameType}\} \to \textsc{Name}\ ev\ n \to \textsc{TypeSchema}\ tv\ n \end{array}$$

3.5.1 Shape Indexing

One of the issues that results from indexing expressions by the set of available type variables is that constraint generation is no longer quite as trivially structurally recursive. For example, take the constraint generation rule for abstraction:

$$\frac{\alpha, \beta, \Delta; \Gamma, (x : \alpha) \mapsto e : \beta \rightsquigarrow C}{\Delta; \Gamma \mapsto \lambda x.\ e : \tau \rightsquigarrow \exists \alpha.\ \exists \beta.\ C \wedge' (\tau \sim (\alpha \to \beta))}\ \textsc{Abs}$$

In this rule, two new type variables are introduced into the set of available variables Δ. This corresponds in our representation to an index $\mathcal{S}\ (\mathcal{S}\ tv)$, however the expression e's index is merely tv. The obvious way to

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{NameType} : \mathrm{Set}\ \textbf{where}} \\
\quad \mathit{Binding} & : & \textsc{NameType} \\
\quad \mathit{Datacon} & : & \mathbb{N} \to \textsc{NameType}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{Name}\ (n : \mathrm{Set})\ :\ \textsc{NameType} \to \mathrm{Set}\ \textbf{where}} \\
\quad N & : & n \to \textsc{Name}\ n\ \mathit{Binding} \\
\quad DC & : & \forall\ \{x\} \to \mathsf{dc}\ x \to \textsc{Name}\ n\ (\mathit{Datacon}\ x)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{Alternative}\ (ev\ tv : \mathrm{Set}) : \textsc{Shape} \to \mathrm{Set}\ \textbf{where}} \\
\quad \xrightarrow{\text{alt}} & : & \forall\{n : \mathbb{N}\}\{\sigma : \textsc{Shape}\} \to \textsc{Name}\ ev\ (\mathit{Datacon}\ n) \to \textsc{Expression}\ (ev\ \oplus\ n)\ tv\ \sigma \to \\
 & & \textsc{Alternative}\ ev\ tv\ (\mathit{Unary}\ \sigma)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{Alternatives}\ (ev\ tv : \mathrm{Set}) : \textsc{Shape} \to \mathrm{Set}\ \textbf{where}} \\
\quad \mathit{esac} & : & \textsc{Alternatives}\ ev\ tv\ \mathit{Nullary} \\
\quad \| & : & \forall\{\sigma_1\ \sigma_2 : \textsc{Shape}\} \to \textsc{Alternative}\ ev\ tv\ \sigma_1 \to \textsc{Alternatives}\ ev\ tv\ \sigma_2 \to \\
 & & \textsc{Alternatives}\ ev\ tv\ (\mathit{Binary}\ \sigma_1\ \sigma_2)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{Expression}\ (ev\ tv : \mathrm{Set}) : \textsc{Shape} \to \mathrm{Set}\ \textbf{where}} \\
\quad \mathit{EVar} & : & \forall\{x : \textsc{NameType}\} \to \textsc{Name}\ ev\ x \to \textsc{Expression}\ ev\ tv\ \mathit{Nullary} \\
\quad \lambda' & : & \forall\{\sigma : \textsc{Shape}\} \to \textsc{Expression}\ (\mathcal{S}\ ev)\ tv\ \sigma \to \textsc{Expression}\ ev\ tv\ (\mathit{Unary}\ \sigma) \\
\quad \mathit{app} & : & \forall\{\sigma_1\ \sigma_2 : \textsc{Shape}\} \to \textsc{Expression}\ ev\ tv\ \sigma_1 \to \textsc{Expression}\ ev\ tv\ \sigma_2 \to \\
 & & \textsc{Expression}\ ev\ tv\ (\mathit{Binary}\ \sigma_1\ \sigma_2) \\
\quad \mathit{let} & : & \forall\{\sigma_1\ \sigma_2 : \textsc{Shape}\} \to \textsc{Expression}\ ev\ tv\ \sigma_1 \to \textsc{Expression}\ (\mathcal{S}\ ev)\ tv\ \sigma_2 \to \\
 & & \textsc{Expression}\ ev\ tv\ (\mathit{Binary}\ \sigma_1\ \sigma_2) \\
\quad \mathit{let}_a & : & \forall\{\sigma_1\ \sigma_2 : \textsc{Shape}\} \to \textsc{Expression}\ ev\ tv\ \sigma_1 \to \mathsf{Type}\ tv \to \textsc{Expression}\ (\mathcal{S}\ ev)\ tv\ \sigma_2 \to \\
 & & \textsc{Expression}\ ev\ tv\ (\mathit{Binary}\ \sigma_1\ \sigma_2) \\
\quad \mathit{let}_{ga} & : & \forall\{\sigma_1\ \sigma_2 : \textsc{Shape}\} \to (n : \mathbb{N}) \to \textsc{Expression}\ ev\ (tv\ \oplus\ n)\ \sigma_1 \to \\
 & & \mathsf{QConstraint}\ (tv\ \oplus\ n) \to \mathsf{Type}\ (tv \oplus n) \to \textsc{Expression}\ (\mathcal{S}\ ev)\ tv\ \sigma_2 \to \\
 & & \textsc{Expression}\ ev\ tv\ (\mathit{Binary}\ \sigma_1\ \sigma_2) \\
\quad \mathit{case} & : & \forall\{\sigma_1\ \sigma_2\} \to \textsc{Expression}\ ev\ tv\ \sigma_1 \to \textsc{Alternatives}\ ev\ tv\ \sigma_2 \to \\
 & & \textsc{Expression}\ ev\ tv\ (\mathit{Binary}\ \sigma_1\ \sigma_2)
\end{array}$$

Fig. 12: The complete Expression representation

remedy this is to simply "up-shift" the de Bruijn indices in e twice using the appropriate substitution functor, however this causes Agda's termination checker to complain — the recursive invocation is no longer on the obviously smaller e; instead it is on the not-so-obviously smaller $\mathit{e\text{-}subst}\ (\mathit{rename}\ (\mathit{suc} \circ \mathit{suc}))\ e$. In order to show termination, we must show that substitution on types (which change the structure of *type* terms) do not change the recursion pattern used by the constraint generation (the structure of *expression* terms). We achieve this using a simple technique: we add a *third* index to the Expression type, which represents the structure or the *shape* of the expression tree (see Fig. 12). The Shape type itself is a straightforward tree structure which is sufficient to encode the recursion pattern of constraint generation:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{Shape} : \mathrm{Set}\ \textbf{where}} \\
\quad \mathit{Nullary} & : & \textsc{Shape} \\
\quad \mathit{Unary} & : & \textsc{Shape} \to \textsc{Shape} \\
\quad \mathit{Binary} & : & \textsc{Shape} \to \textsc{Shape} \to \textsc{Shape}
\end{array}$$

With this addition, the type of the substitution functor for expressions now becomes:

$$\mathit{e\text{-}subst} : \forall\{\sigma : \textsc{Shape}\}\ \{ev\ \alpha\ \beta : \mathrm{Set}\} \to (\alpha \to \mathsf{Type}\ \beta) \to \textsc{Expression}\ ev\ \alpha\ \sigma \to \textsc{Expression}\ ev\ \beta\ \sigma$$

The fact that the shape of the expression is not changed by type substitution is now made clear in the type of the type substitution operation. In constraint generation, recursion once again becomes structural, not on the expressions themselves, but on their shape parameters.

3.6 Constraint Generator and Solver

Now that we have established our choice of representation for terms and types, we must still choose a representation for the *computations* themselves, specifically constraint generation, and solving.

Broadly speaking, there are two main ways to encode this computation. The first is to directly encode the computation as an Agda function (using constraint generation as an example):

$$\begin{aligned}
\mathit{genConstraint} : \{\mathit{ev}\ \mathit{tv} : \mathrm{Set}\}\ \{\sigma : \textsc{Shape}\} \\
&\to \mathit{Environment}\ \mathit{ev}\ \mathit{tv} \\
&\to \textsc{Expression}\ \mathit{ev}\ \mathit{tv}\ \sigma \\
&\to \mathsf{Type}\ \mathit{tv} \\
&\to \textsc{Constraint}\ \mathit{tv} \\
\ldots
\end{aligned}$$

This is certainly the most straightforward encoding of the algorithm, and was the method we had used during development of our formalisation. Ultimately, we chose not to use this simple representation, however, opting instead for a *propositional* approach, defining a propositional data type for the constraint generation judgement:

$$\begin{aligned}
&\textbf{syntax}\ \textsc{ConstraintGen}\ \Gamma\ \tau\ e\ C = \Gamma \mapsto e : \tau \rightsquigarrow C \\
&\textbf{data}\ \textsc{ConstraintGen}\ \{\mathit{ev}\ \mathit{tv} : \mathrm{Set}\}\ (\Gamma : \mathit{Environment}\ \mathit{ev}\ \mathit{tv})\ (\tau : \mathsf{Type}\ \mathit{tv}) : \\
&\qquad \{\sigma : \textsc{Shape}\} \to \textsc{Expression}\ \mathit{ev}\ \mathit{tv}\ \sigma \to \textsc{Constraint}\ \mathit{tv}\ \mathit{Hi} \to \mathrm{Set}\ \textbf{where} \\
&\quad \ldots
\end{aligned}$$

This type contains constructors corresponding to each rule in the constraint generation (see Fig. 3). Then, to provide an algorithmic interpretation of these rules, and therefore provide a way to actually *run* our constraint generator and infer types, we write a function that generates a proof of this proposition given the requisite inputs[18]:

$$\begin{aligned}
\mathit{genConstraint}' \ :\ & \{\mathit{ev}\ \mathit{tv} : \mathrm{Set}\}\ \{\sigma : \textsc{Shape}\}\ (\Gamma : \mathit{Environment}\ \mathit{ev}\ \mathit{tv})\ (e : \textsc{Expression}\ \mathit{ev}\ \mathit{tv}\ \sigma)\ (\tau : \mathsf{Type}\ \mathit{tv}) \\
\to\ & \exists\ (\lambda\ C \to \Gamma \mapsto e : \tau \rightsquigarrow C)
\end{aligned}$$

This approach brings a number of advantages. For one, it brings a substantial performance improvement — the constraint generator takes one tenth the time and one fifth the memory to type check. It also gives us an Agda proposition exactly the same in form to the rules presented in the previous chapter. Not only a nice aesthetic consequence, this also liberates us from certain petty annoyances that arise from encoding the algorithm directly. For example, the solver is partial — not all constraints are satisfiable. In a direct functional encoding, this would necessitate `Maybe` types or similar mechanisms to deal with partiality. The propositional encoding of the solver, however, is only ever constructible in the happy case; the partiality is relegated to the algorithmic interpretation of the proposition.

If we desire to prove things about these operations (for example, inference soundness and principality), then the propositional encoding of these algorithms is much more preferable. Instead of evaluating expressions and pattern matching on their results, we can more directly match on the various rules used to justify the proposition. This makes proofs done on paper about these systems much more straightforward to transfer to our formalisation.

Another advantage of the propositional encoding comes from the solver. Observe the rule for the solver infrastructure:

$$\frac{\mathcal{Q}; Q_g; \bar{\alpha} \overset{\text{simp}}{\mapsto} Q \rightsquigarrow Q_r; \theta \qquad \forall((\exists^I \bar{\beta}_i.\ Q_i \supset \mathcal{C}_i) \in I).\ \mathcal{Q}; Q_g \wedge Q_r \wedge Q_i; \bar{\beta}_i \overset{\text{solv}}{\mapsto} \mathcal{C}_i \rightsquigarrow \epsilon; \theta_i}{\mathcal{Q}; Q_g; \bar{\alpha} \overset{\text{solv}}{\mapsto} Q \cdot I \rightsquigarrow Q_r; \theta}$$

In particular, note that each implication constraint body must resolve to ϵ. In a propositional encoding, this is a non-issue, but an algorithmic encoding must check the return value of the solver function given the implication constraint, to *ensure* that the result is ϵ. As $\mathsf{QConstraint}$s are part of the parameter and therefore abstract, we cannot pattern match to make these assurances, and must therefore demand a proof of the decidability of equality to ϵ in the X constraint:

$$\textbf{field}\ \mathsf{dec}\text{-}\epsilon : (x : \mathsf{QConstraint}) \to \textsc{Dec}\ (x \equiv \epsilon)$$

[18] Interestingly, as this function must be total and terminating, this can be viewed as a proof of the decidability of the ConstraintGen proposition

Where Dec is defined (in the standard library) as follows:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{data}\ \textsc{Dec}\ (P : \mathrm{Set}) : \mathrm{Set}\ \textbf{where}} \\ \quad \textit{yes} & : & P \rightarrow \textsc{Dec}\ P \\ \quad \textit{no} & : & (P \rightarrow \bot) \rightarrow \textsc{Dec}\ P \end{array}$$

By adopting a propositional encoding for the solver, we do not eliminate this issue, but at least we are able to confine it merely to the algorithmic interpretation of the proposition rather than have such nonsense pollute the rules themselves.

4 Simple Instantiation

This chapter describes a simple instantiation of the X parameter of the $\textsc{OutsideIn}(X)$ system. We have formalised this instantiation in Agda and used it to infer types for basic programs.

By developing this simple instantiation, we not only provide a way to actually *use* the type checker we have formalised, but we also ensure that it is in fact *possible* to instantiate. Otherwise, it would be possible that the requirements in the X parameter were too onerous to actually be instantiable[19].

Our instantiation of QConstraint is the smallest that meets the requirements of the X parameter. It consists only of ϵ, constraint conjunction and type equality constraints:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{QConstraint}\ (x : \text{Set}) : \text{Set}\ \textbf{where}} \\
\quad \sim & : & \textsc{Type}\ x \rightarrow \textsc{Type}\ x \rightarrow \textsc{QConstraint}\ x \\
\quad \wedge & : & \textsc{QConstraint}\ x \rightarrow \textsc{QConstraint}\ x \rightarrow \textsc{QConstraint}\ x \\
\quad \epsilon & : & \textsc{QConstraint}\ x
\end{array}$$

$\textsc{Type}$ consists of type variables (or constructors), type application and function types — the typical assortment of types from System F.

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \textsc{Type}\ (x : \text{Set}) : \text{Set}\ \textbf{where}} \\
\quad \mathit{Var} & : & n \rightarrow \textsc{Type}\ n \\
\quad \rightarrow' & : & \textsc{Type}\ n \rightarrow \textsc{Type}\ n \rightarrow \textsc{Type}\ n \\
\quad \mathit{app} & : & \textsc{Type}\ n \rightarrow \textsc{Type}\ n \rightarrow \textsc{Type}\ n
\end{array}$$

$\textsc{Type}$ is also a monad, as required by the X parameter, but this proof is entirely uneventful and uninteresting, as is the proof of the functor laws for the *Q-subst* functor. Indeed, the only truly interesting component of the simple instantiation is the simplifier.

The simplest possible simplifier is simply the no-op:

$$\mathcal{Q}\ ;\ Q_g\ ;\ \overline{\alpha_{\text{tch}}} \mapsto Q_w \rightsquigarrow Q_w\ ;\ \mathit{unit}_{\mathsf{Type}}$$

While this simplifier meets the simplifier soundness and principality criteria in Figure 2, it is not a particularly interesting simplifier to work with. Instead, we define a simplifier that is actually capable of solving constraints.

Broadly speaking, the simplifier is divided into two main components:

1. **Unification**, for solving equality constraints.
2. **Solving**, which uses the results from unification to solve constraint terms.

4.1 Unification

Unification is the process of producing a substitution θ given two terms τ_1 and τ_2 such that $\theta\tau_1 \equiv \theta\tau_2$. Unification for first-order terms is decidable via Robinson's algorithm (Robinson, 1965), but this has a nontrivial termination argument, based on the fact that each variable substituted reduces the number of available variables in the term. This gives a termination measure, but it is not one that is immediately visible as a structural recursion. This is problematic in Agda, which mandates that all functions be structurally recursive.

McBride more recently demonstrated a structurally recursive presentation of first-order unification in a dependently typed setting, indexing terms by the number of available unification variables (McBride, 2003). For the purposes of our unification, we shamelessly reuse his presentation with little modification.

One slight difference is that our $\textsc{Type}$ terms do not come indexed by the number of available variables they contain. Rather, they come in the form $\textsc{Type}\ (\tau \oplus n)$, with skolem variables and constructor names represented by the type τ, and n unification variables available. We define our unification algorithm in terms of a special

[19] As a most extreme example: if we had, by some accident, required a proof of $\bot$ in our X parameter, this would not only be impossible to instantiate, it would also render any proof about the system entirely useless!

$$\boxed{\mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} Q_w \leadsto Q_r \;;\; \theta}$$

$$\frac{}{\mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} \epsilon \leadsto \epsilon \;;\; \mathit{unit}_{\mathsf{Type}}}\ \textsc{Empty}$$

$$\frac{\mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} Q_1 \leadsto Q_1' \;;\; \theta_1 \qquad \mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} \theta_1 Q_2 \leadsto Q_2' \;;\; \theta_2}{\mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} Q_1 \wedge Q_2 \leadsto \theta_2 Q_1' \wedge Q_2' \;;\; \theta_2 \circ_{\mathsf{Type}} \theta_1}\ \textsc{Conj}$$

$$\frac{\tau_1 \text{ mgu } \tau_2 \leadsto \theta}{\mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} \tau_1 \sim \tau_2 \leadsto \epsilon \;;\; \theta}\ \textsc{Mgu} \qquad \frac{Q_g \overset{?}{\mapsto} \tau_1 \sim \tau_2}{\mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} \tau_1 \sim \tau_2 \leadsto \epsilon \;;\; \mathit{unit}_{\mathsf{Type}}}\ \textsc{Entail}$$

$$\frac{}{\mathcal{Q} \;;\; Q_g \overset{\text{simp}}{\mapsto} \tau_1 \sim \tau_2 \leadsto \tau_1 \sim \tau_2 \;;\; \mathit{unit}_{\mathsf{Type}}}\ \textsc{GiveUp}$$

$$\boxed{Q_g \overset{?}{\mapsto} Q_w}$$

$$\frac{}{Q \overset{?}{\mapsto} Q}\ \textsc{Refl} \qquad \frac{Q_1 \overset{?}{\mapsto} Q}{Q_1 \wedge Q_2 \overset{?}{\mapsto} Q}\ \textsc{ConjE}_1 \qquad \frac{Q_2 \overset{?}{\mapsto} Q}{Q_1 \wedge Q_2 \overset{?}{\mapsto} Q}\ \textsc{ConjE}_2$$

$$\boxed{\tau_1 \text{ mgu } \tau_2 \leadsto \theta}$$

(first order unification)

Fig. 13: Simple instantiation of the simplifier

type used for names, which explicitly separates skolem from unification variables — after all, they are treated very differently when performing unification:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{data}\ \textsc{SName}\ (\mathit{sk} : \mathrm{Set})\ (\mathit{un} : \mathbb{N}) : \mathrm{Set}\ \textbf{where}} \\ \quad \mathit{unification} & : & \textsc{Fin}\ \mathit{un} \to \textsc{SName}\ \mathit{sk}\ \mathit{un} \\ \quad \mathit{rigid} & : & \mathit{sk} \to \textsc{SName}\ \mathit{sk}\ \mathit{un} \end{array}$$

Then, we define the straightforward isomorphism between the type used for names in the simplifier, and this special SName type used in the unification:

$$\mathit{iso}_1 : \forall \{m : \mathbb{N}\}\{t : \mathrm{Set}\} \to t \oplus m \to \textsc{SName}\ t\ m$$
$$\mathit{iso}_2 : \forall \{m : \mathbb{N}\}\{t : \mathrm{Set}\} \to \textsc{SName}\ t\ m \to t \oplus m$$

This isomorphism can be applied to types and constraints using the standard *rename* functors, which allows us to use a convenient representation for unification without affecting any other component of the instantiation or of the system.

As this representation trivially ensures that the domain of any substitution is restricted only to the unification variables $\overline{\alpha_{\text{tch}}}$, we omit mention of $\overline{\alpha_{\text{tch}}}$ in Figure 13.

4.2 Solving

The overall simplifier has to deal with the three possible constraint forms (see Fig. 13). Empty ϵ constraints resolve trivially to the identity substitution, conjunctions resolve to a composition of the substitutions resulting from each of the two conjuncts, and type equality resolves to the most general unifier of the two types. There are two other rules for type equality: Entail, which searches the given context for an identical constraint to the wanted equality; and GiveUp, which simply gives up on solving the constraint entirely, returning it as a residual constraint. The presence of these rules makes the system not syntax-directed, and therefore it does not directly correspond to a deterministic algorithm. We resolve this nondeterminism by introducing an ordering on the rules: Mgu is attempted first, then Entail, then GiveUp.

Another problem presents itself when one attempts to encode this simplifier in Agda. In the rule Conj, the first substitution θ is applied to the second conjunct Q_2, and recursion occurs on the substituted constraint θQ_2. This recursion is *non-structural* — the dreaded termination checker has once again reared its head.

In order to convince Agda that applying a type substitution to a constraint does not affect the shape of the constraint, we employ a similar tactic to the technique used for representing Expressions previously. We shall define a new form of constraint, indexed by its *shape*:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \text{SConstraint}\ (x : \text{Set}) : \text{Shape} \to \text{Set}\ \textbf{where}} \\
\sim & : & \text{Type}\ x \to \text{Type}\ x \to \text{SConstraint}\ x\ \textit{Nullary} \\
\wedge & : & \forall\{s_1\ s_2\} \to \text{SConstraint}\ x\ r_1 \to \text{SConstraint}\ x\ r_2 \to \text{SConstraint}\ x\ (\textit{Binary}\ s_1\ s_2) \\
\epsilon & : & \text{SConstraint}\ x\ \textit{Nullary}
\end{array}
$$

With this indexed data type, substitution now has the following type:

$$\textit{SC-subst} : \forall\{\alpha\ \beta\}\{s\} \to (\alpha \to \text{Type}\ \beta) \to \textit{SConstraint}\ \alpha\ s \to \textit{SConstraint}\ \beta\ s$$

As with Expression, we can now glean from the type that the *shape* of the constraint term does not change with substitution. Therefore, structural recursion can occur on the Shape index of the SConstraint term, rather than on the term itself. As SConstraint and QConstraint are structurally identical, it is trivial to convert between them.

4.3 Results

We have used this simple instantiation to successfully infer types, including polymorphic types and types involving GADTs. In particular, recall the example used earlier to illustrate the necessity of removing `let`-generalisation:

$$
\begin{array}{lll}
\textbf{data}\ \ \textit{IntOrBool} & :: & * \to *\ \textbf{where} \\
\quad\text{IsInt} & :: & (\tau \sim \textit{Int}) \Rightarrow \textit{IntOrBool}\ \tau \\
\quad\text{IsBool} & :: & (\tau \sim \textit{Bool}) \Rightarrow \textit{IntOrBool}\ \tau
\end{array}
$$

$$
\begin{array}{l}
f :: \textit{IntOrBool}\ \alpha \to \alpha \to \textit{Bool} \\
f\ x\ y = \textbf{let}\ g\ z\ = \textit{not}\ y\ \textbf{in} \\
\qquad\qquad \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\quad \textit{IsInt} \ \to\ \text{True} \\
\qquad\qquad\quad \textit{IsBool} \ \to\ g\ ()
\end{array}
$$

This expression rightly fails to type-check in our simple instantiation, however if we add a type signature to g:

$$
\begin{array}{l}
f :: \textit{IntOrBool}\ \alpha \to \alpha \to \textit{Bool} \\
f\ x\ y = \textbf{let}\ g : \forall\beta.\ (\alpha \sim \textit{Bool}) \to \beta \to \textit{Bool} \\
\qquad\qquad\quad g\ z\ = \textit{not}\ y\ \textbf{in} \\
\qquad\qquad\quad \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad \textit{IsInt} \ \to\ \text{True} \\
\qquad\qquad\qquad \textit{IsBool} \ \to\ g\ ()
\end{array}
$$

The program now type-checks in our simple instantiation of OutsideIn(X), as desired.

5 Conclusions and Future Work

Our formalisation comes to approximately three thousand lines of Agda definitions and proofs, excluding comments, which makes it one of the largest single formalisations in Agda to date, approximately one fifth the size of the entire Agda standard library. It takes approximately 46 seconds to type check, using the latest development version of Agda 2.3.2 on a mid-2011 MacBook Pro. Individual modules take no longer than five seconds to check.

During our development, constraint generation and other phases of the algorithm were written predominantly in a direct, functional style, rather than the final propositional encoding which we present here. Constraint generation, written this way, took approximately five minutes to type check, and required approximately six gigabytes of RAM — just for the constraint generation module alone. Exactly what causes this is still unclear, however we have raised the issue with the Agda developers.

While in development, we attempted to make extensive use of several experimental Agda features, such as instance arguments (Devriese & Piessens, 2011), however this frequently resulted in strangely unsolved metavariables, confusing error messages, or worse, internal errors from Agda's type checker.

One of our goals in this endeavour is to show Agda's readiness for type systems work. Despite all of the above problems, we believe we have succeeded in this respect. Agda has, for the most part, provided a very expressive language for us to encode our formalisation. Certain tricks, such as our encoding of expression names or our indexing of type terms, would be highly uncommon in proof assistants such as Coq and impossible in theorem provers such as Isabelle. On the other hand, Agda requires that all termination arguments be phrased in terms of structural recursion, which did result in some minor contortions to our formalisation that would not be necessary if a separate termination proof could be provided. Conor McBride has made the argument that such contortions are merely exposing pre-existing structure of our algorithms that was previously hidden, and therefore a structurally recursive presentation is clearer than a presentation with a separate termination proof (McBride, 2003). We are not certain whether we agree with this argument, but offer it as justification for the abovementioned contortions.

Agda is most definitely a programming language *first* and a proof assistant *second*. For this reason, it could be argued that we have been playing to Agda's strengths — having encoded the structures, definitions and algorithms of the system, we have formalised the part of the system to which Agda is best suited. It remains to be seen whether Agda is suitable for serious proof work (see section 5.1 for a discussion of a possible soundness proof for our formalisation).

Another goal of our research was to provide a base upon which other type system formalisations can be developed, and to create a testbed for experimenting with OutsideIn(X) on solid formal ground. By including a simple instantiation of the system, we have demonstrated the capability of our formalisation to represent real type systems and infer types for real programs. An obvious direction for future work is to instantiate X for all of Haskell, a prospect we discuss in section 5.2, as well as expanding the simple instantiation to include proofs of the simplifier soundness and principality conditions.

5.1 Proofs

We have completed a preliminary investigation into a proof of *soundness* for the inference system we have presented. *Soundness* here refers the notion that if a particular type τ is inferred for an expression e, then there exists a *proof* according to some natural, permissive typing rules that e is well typed by τ. Also of interest is a *principality* proof, which suggests that any inferred type of an expression e is the principal type for e. The original OutsideIn(X) paper presents typing rules as well as a proof of soundness and principality for their system, assuming simplifier soundness and guess-freedom conditions. A number of obstacles make converting those proofs to our formalisation less than straightforward:

1. The proofs presented in the OutsideIn(X) paper are not very detailed, mostly due to space constraints, and as a result some steps in the proof are not very clear. For example, both proofs mention induction, however the exact structure of the induction is not clarified; and in both proofs, several cases are dismissed as similar to a previous case, without clarification as to exactly what is different or how they are similar.

2. Some properties of entailment upon which the soundness proof relies are not even formalised in the entailment relation presented in the paper. For example, it is not required that ϵ is entailed by any context, nor is conjunction elimination included in the original presentation of entailment. Nevertheless, these properties are relied upon by the soundness proof.
3. Our formalisation introduces a few new layers of indirection between constraint generation and constraint solving (such as the prenexer, separator and so on). This added indirection means that the constraint produced by the constraint generator and the constraint given to the solver are not the same, which makes reasoning more difficult.
4. The original proof takes certain properties of conjunctions as automatic and invisible; specifically associativity, commutativity, and identity with ϵ. This means that goals might be perfectly provable but not fitting to the exact structure of the constraints required, which can make proving much more difficult.

The first two points here mostly concern problems with the original OutsideIn(X) proof. These problems are not insurmountable, and indeed they are to be expected — all proofs become sloppy when viewed through the harsh, clear lens of a proof assistant. The final two points, however, reveal some weaknesses in Agda's approach to theorem proving. It is common to find oneself *encumbered by structure* in languages such as Agda, where proofs of even simple properties can be complicated by the wrong choice of representation. In Isabelle, for example, correspondences across layers of indirection and simple lemmas about entailment (such as commutativity of conjunction) can be added to an automatic simplifier that would make these problems trivially soluble. Agda, on the other hand, places on us the onerous burden of manually rewriting terms to be of the correct structure, and manually transporting properties across isomorphisms.

While these obstacles are irritating, they are not impossible to overcome. Therefore, our subsequent research effort shall be devoted to developing a soundness proof for our formalisation.

5.2 Instantiating X for Haskell

Another obvious progression from this point is to investigate an instantiation of the X parameter with a sophisticated type system that supports the many extensions available in Haskell, such as type classes (and their corresponding top-level axiom schemes), type families, and so on. A surface evaluation reveals that this would likely be quite difficult, chiefly because the simplifier for this Haskell-like type system does not have a straightforward termination measure — Indeed, none is presented in the original OutsideIn(X) paper, which presents this simplifier as a sequence of rewrite rules. The GHC implementation of the simplifier also has no obvious termination argument. While a proof of termination could no doubt be found (the system does, after all, appear to terminate!), it may well be difficult to phrase such a sophisticated simplifier as a structural recursion.

One method towards circumventing this problem (which does not involve determining a complicated termination argument for the rewrite system) is to simply pass an arbitrary large natural number to the rewrite system which reduces this number by one on each rule application, failing if the number reaches zero. A proof of termination would then be rephrased as a proof that there exists a number for any term which, when given to this system along with the term, will result in an expression in normal form (i.e. not fail). By rephrasing it this way, Agda will view the rewrite system as structurally recursive in the natural number, and a separate proof of termination can be provided[20].

5.3 Summary of Contributions

- A more rigorous and clear formulation of the OutsideIn(X) inference system.
- An Agda formalisation of this system, instantiable for real type systems, that can be used for experimentation and development of new type system extensions on solid formal ground.
- A simple example instantiation of the above Agda formalisation, which can be used to infer types for basic programs, and can also serve as a basis for more sophisticated instantiations.
- A demonstration of a variety of representational techniques for names and terms, suitable for similar work in both dependently-typed languages like Agda, and almost-dependently-typed languages like Haskell.

[20] Or simply assumed, if no proof is found.

Through our work on OutsideIn(X), we have shown that Agda gives us through its type system a very useful and powerful language to express structure and maintain invariants about our data. The addition of a soundness and principality proof, an obvious direction for future work, would make our formalisation highly compelling for those seeking to extend or experiment with Haskell's type inference system. Even without such a proof, our formalisation already serves as a platform for experimentation, as it is the only implementation of OutsideIn(X) that is actually abstracted over its X parameter. Ultimately, this project has brought us one small step closer to a rigorously formalised type system for GHC Haskell.

Acknowledgements. The author would like to thank the members of the Agda community, specifically Arseniy Alekseyev, Conor McBride, Daniel Peebles, Andrea Vezzosi and others, all of whom were of great assistance when learning Agda and developing this formalisation. Edward Kmett and Ashley Yakely also offered their advice and assistance during the development of the category theory components of the formalisation. Thanks are also due to Manuel Chakravarty, who supervised this project, and the entire PLS group at UNSW.

On a personal note, I would like to thank my partner Tran for her support and understanding in this busy year.

References

1 Thorsten Altenkirch. (1999). Extensional Equality in Intensional Type Theory. *Pages 412–412 of: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science.* Washington, DC, USA: IEEE Computer Society.

2 Françoise Bellegard, and James Hook. (1994). Substitution: a Formal Methods Case Study Using Monads and Transformations. *Science of computer programming*, **23**(2-3), 287–311.

3 Richard S. Bird, and Ross Paterson. (1999). de Bruijn Notation as a Nested Datatype. *Journal of functional programming*, **9**(1), 77–91.

4 Nils Danielsson, and Ulf Norell. (2011). Parsing Mixfix Operators. *Pages 80–99 of: Implementation and Application of Functional Languages.* Springer Berlin / Heidelberg.

5 Nicolaas Govert de Bruijn. (1972). Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem. *Indagationes mathematicae (elsevier)*, **34**, 381–392.

6 Dominique Devriese, and Frank Piessens. (2011). On the bright side of type classes: instance arguments in Agda. *Pages 143–155 of: ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming.* New York, New York, USA: ACM.

7 Catherine Dubois. (2000). Proving ML Type Soundness Within Coq. *Pages 126–144 of: TPHOLs '00: Proceedings of "Theorem Proving with Higher Order Logics" 2000.* Springer.

8 Catherine Dubois, and Valérie Ménissier-Morain. (1999). Certification of a type inference tool for ML: Damas-Milner within Coq. *Journal of automated reasoning*, **23**, 3–4.

9 W.A. Howard. (1980). *The Formulae-as-Types Notion of Construction.* Letters To H.B. Curry: Essays on Combinatory Logic.

10 Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. (2010). Fun with Type Functions. *Pages 301–331 of: Reflections on the Work of C.A.R. Hoare.* London: Springer London.

11 Simon Marlow (Ed.). (2010). *Haskell 2010 Language Report.* Tech. rept.

12 Per Martin-Löf. (1984). *Intuitionistic Type Theory.* Bibliopolis.

13 Conor McBride. (2003). First-Order Unification by Structural Recursion. *Journal of functional programming*, **13**(6), 1061–1075.

14 Conor McBride, and James McKinna. (2004). The View From The Left. *Journal of functional programming*, **14**(1), 69–111.

15 Robin Milner. (1978). A Theory of Type Polymorphism in Programming. *Journal of computer and system sciences*, **17**, 348–375.

16 Peter Morris, Thorsten Altenkirch, and Conor McBride. (2004). Exploring the Regular Tree Types. *Page 30 of: Types for Proofs and Programs.* SpringerVerlag.

17 Wolfgang Naraschewski, and Tobias Nipkow. (1999). Type Inference Verified: Algorithm W in Isabelle/HOL. *Journal of automated reasoning*, **23**(3), 299–318.

18 Ulf Norell. 2008 (May). Dependently Typed Programming In Agda. *AFP 08: Sixth International Summer School on Advanced Functional Programming.*

19 Martin Odersky, Martin Sulzmann, and Martin Wehr. (1997). Type Inference with Constrained Types. *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL).*

20 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. (2007). Practical Type Inference for Arbitrary-Rank Types. *Journal of functional programming*, **17**(01), 1–82.

21 François Pottier, and Didier Rémy. (2005). Essence of ML Type Inference. Benjamin Pierce (ed), *Advanced Topics in Types and Programming Languages.* Cambridge, Massachuetts: MIT Press.

22 Nicolas Pouillard. (2011). Nameless, Painless. *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming.* ACM Request Permissions.

23 John Alan Robinson. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the acm*, **12**(1), 23–41.

24 Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. (2009). Complete and Decidable Type Inference for GADTs. *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming.* ACM Request Permissions.

25 Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. (2010). Let Should Not Be Generalized. *TLDI '10: Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation.* ACM Request Permissions.

26 Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Suzmann. (2011). OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of functional programming*, **21**(4-5), 333–412.

27 Philip Wadler. (1989). Theorems for free! *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture (FPCA '89).*

28 Brent Yorgey, Stephanie Weirich, Simon Peyton Jones, Julian Cretin, Dimitrios Vytiniotis, and José Pedro Magalhães. (2012). Giving Haskell a Promotion. *Pages 53–66 of: Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation.* New York, NY, USA: ACM.