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

School of Computer Science and Engineering
University of New South Wales

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY - AUSTRALIA

MAY 20TH 2012
Abstract

GHC, a 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 rigor 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

1 Introduction 4
  1.1 Related Work 4
  1.2 Why mechanise OutsideIn(X)? 4
  1.3 Why Agda? 5
  1.4 A Brief Introduction to Agda 5
  1.5 Overview 6

2 Making OutsideIn(X) Rigorous 7
  2.1 Regarding let-generalisation 7
  2.2 Phases of Inference 8
  2.3 The Extended Constraint Language 9
  2.4 Fresh Variables 10
  2.5 The Prenexer 12
  2.6 Solver and Separator 13
  2.7 Top Level Rules 14

3 Encoding Types and Terms 15
  3.1 Names 15
  3.2 Indexed Terms 16
  3.3 Nested Data Types 17
  3.4 Separator and Prenexer 21
  3.5 Expressions 22
  3.6 Constraint Generator and Solver 24
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 is intended 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 Naraschewski have formalised Milner’s original $W$ 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 rigor 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.

1.3 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 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. 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.

```haskell
data EqOrShow :: * -> * where
  IsEq :: (Eq τ) => EqOrShow τ
  IsShow :: (Show τ) => EqOrShow τ
```

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

```haskell
f :: EqOrShow α -> α -> Either String Bool
f IsEq x = Right (x \equiv x)
f IsShow x = Left (show x)
```

In this case, the local assumption \( \text{Eq } \alpha \) allows the first alternative to type check, while the local assumption \( \text{Show } \alpha \) 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 constraint 4

```haskell
data IntOrBool :: * -> * where
  IsInt :: (τ ~ Int) => IntOrBool τ
  IsBool :: (τ ~ Bool) => IntOrBool τ
```

A function which pattern matches on this GADT must obviously localise each constraint assumption to each alternative — it is impossible for \( \alpha \sim \text{Bool} \) and \( \alpha \sim \text{Int} \) unless \( \text{Int} \sim \text{Bool} \), 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 \( \text{Int} \) and \( \text{Bool} \) 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:

\[
\begin{align*}
  f &:: \text{IntOrBool} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \text{Bool} \\
  f \ x \ y &= \text{let} \ g \ z = \text{not} \ y \ \text{in} \\
  \text{case} \ x \ \text{of} \\
  \text{IsInt} &\Rightarrow \text{True} \\
  \text{IsBool} &\Rightarrow g ()
\end{align*}
\]

Most programmers would expect the binding \( g \) to give a type error, as it requires \( y \) to be of type \( \text{Bool} \), before that has been established via pattern matching on \( x \). A principal type does exist, however, for \( g \):

\[
g :: \forall \beta. (\alpha \sim \text{Bool}) \Rightarrow \beta \Rightarrow \text{Bool}
\]

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

The authors of \textsc{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 \( W \) algorithm \cite{Milner1978}; 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 \cite{PeytonJones2007}. 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 \textsc{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) \text{let}-binding \text{let} \( x = y \ \text{in} \ e \) is equivalent to \((\lambda x. e) \ y\). While a cherished feature of most type inference algorithms since the original \( W \), \text{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 \texttt{base} Haskell library consisting of 94,954 lines after this change was made \cite{Vytiniotis2010}.

### 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. \textsc{OutsideIn}(\( X \)), however, presents inference as two separate phases:

1. \textit{Generate} constraints (according to a set of syntax-directed rules), combining them into one large overall constraint.
2. \textit{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 \textsc{OutsideIn}(\( X \)) (at least in theory) keeps them distinct. Our formal work is derived from \textsc{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.

\footnote{Note that \( \text{not} :: \text{Bool} \Rightarrow \text{Bool} \)}
Constraint generation in \textsc{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 \textit{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 \textit{fig. 2}). This parameterisation of the system is similar to HM(X), the parameterised extension of ML’s type inference pre-

er \textit{fig. 1}: Our updated syntax, slightly extended from \textsc{OutsideIn}(X)

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
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 & $\in$ \(T\) \\
Expressions & $\nu ::= K | x$ \\
& \hspace{1cm} \mid \text{case } e \text{ of } \{K \ x \rightarrow e\}$ \\
& \hspace{1cm} \mid \text{let } x = e_1 \text{ in } e_2 | \text{let } x :: = e_1 \text{ in } e_2 \\
Type schemes & $\sigma ::= \forall \alpha. Q \Rightarrow \tau$ \\
Programs & $\text{prog} ::= \forall \sigma = e, \text{prog} | f = e, \text{prog} | e$ \\
Constraints* & $Q ::= e | Q_1 \land Q_2 | \tau_1 \Rightarrow \tau_2 | \cdots$ \\
Extended constraints & $\bar{C} ::= \{Q_1 \land \bar{C}_1 | \bar{C}_2 \land \bar{C}_3 | \bar{C}_4 \Rightarrow \tau| \bar{C}_5\}$ \\
Monotypes* & $\bar{\tau} ::= iv | T \bar{\tau} | \tau_1 \Rightarrow \tau_2 | \cdots$ \\
Environments & $\Gamma ::= e | (\nu : \sigma), \Gamma$ \\
Axiom Schema* & $Q ::= \cdots$
\end{tabular}
\caption{Our updated syntax, slightly extended from \textsc{OutsideIn}(X)}
\end{table}

2.3 The Extended Constraint Language

In order to deal with local assumptions, \textsc{OutsideIn}(X) extends \(Q\), the original constraint language in \(X\), to an \textit{algorithmic} constraint language \(C\). \(C\) is just \(Q\) with an additional form, \(\exists \beta. (Q \Rightarrow C)\), where \(Q\) is a local assumption, \(C\) is a constraint, and \(\beta\) are the only variables that can be unified while solving the constraint \(Q \Rightarrow C\). 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 \textsc{OutsideIn}(X) in several respects. One minor change is that we make the separation between the languages \(C\), \(Q\) and \(Q\) much more clear. In particular, we add a new form of conjunction to the extended language \(C\), written \(\phi \land \psi\), as the conjunction inherited from \(Q\) can, technically, only contain \(Q\)-constraints. In addition, we have changed \(Q\), the language of axiom schema, so that it no longer includes all of \(Q\). Instead, we make no assumptions about the forms that \(Q\) schema may take, and have reformulated the entailment relation to require both a \(Q\)-constraint and a \(Q\)-constraint as context (see \textit{fig. 2}). We have also added to the entailment relation requirements for conjunction elimination rules as well as a rule to deal with \(e\)-constraints. These rules were inexplicably absent from the original presentation, despite being implicitly used repeatedly in the soundness proof presented in the same work.

\footnote{or the more rigorous formalisation presented by Pottier and Rémy in their chapter of \textit{Advanced Types and Programming Languages}}

\begin{thebibliography}{99}

\bibitem{Pottier & Remy, 2005} Pottier \& Remy, 2005
\end{thebibliography}
part of the parameter $X$, subject to the following requirements:

Tautology \[ \mathcal{Q}; Q \models \epsilon \] (R₁)

Reflexivity \[ \mathcal{Q}; Q \models Q \] (R₂)

Transitivity \[ \mathcal{Q}; Q₁ \models Q₂ \text{ and } \mathcal{Q}; Q₂ \models Q₃ \text{ implies } \mathcal{Q}; Q₁ \models Q₃ \] (R₃)

Substitution \[ \mathcal{Q}; Q₁ \models Q₂ \text{ implies } \theta; \mathcal{Q}; \theta Q₁ \models \theta Q₂ \text{ where } \theta \text{ is a type substitution} \] (R₄)

Conjunction intro. \[ \mathcal{Q}; Q \models Q₁ \text{ and } \mathcal{Q}; Q \models Q₂ \text{ implies } \mathcal{Q}; Q \models Q₁ \land Q₂ \] (R₅)

Conjunction elim. \[ \mathcal{Q}; Q₁ \land Q₂ \models Q₁ \] (R₆)

\[ \mathcal{Q}; Q₁ \land Q₂ \models Q₂ \] (R₇)

Type eq. reflexivity \[ \mathcal{Q}; Q \models \tau \sim \tau \] (R₈)

Type eq. symmetry \[ \mathcal{Q}; Q \models \tau₁ \sim \tau₂ \text{ implies } \mathcal{Q}; Q \models \tau₂ \sim \tau₁ \] (R₉)

Type eq. transitivity \[ \mathcal{Q}; Q \models \tau₁ \sim \tau₂ \text{ and } \mathcal{Q}; Q \models \tau₂ \sim \tau₃ \text{ implies } \mathcal{Q}; Q \models \tau₁ \sim \tau₃ \] (R₁₀)

Type eq. substitutivity \[ \mathcal{Q}; Q \models \tau₁ \sim \tau₂ \text{ implies } \mathcal{Q}; Q \models [α \mapsto \tau₁] \tau \sim [α \mapsto \tau₂] \tau \] (R₁₁)

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

Touchable-aware: \[ \mathcal{Q}; Q_{\text{given}}; \alpha \mathcal{G}_{\text{th}} \models Q_{\text{wanted}} \sim Q_{\text{residual}}; \theta \]

Soundness: \[ \mathcal{Q}; Q_{g} \models Q_{w} \sim Q_{i}; \theta \text{ implies } \mathcal{Q}; Q_{g} \wedge Q_{r} \models \theta Q_{w} \]

Principality: (Guess-freedom) \[ \mathcal{Q}; Q_{g} \models Q_{w} \sim Q_{i}; \theta \text{ implies } \mathcal{Q}; Q_{g} \wedge Q_{w} \models Q_{r} \wedge \mathcal{L}_{\theta} \]

where \( \mathcal{L}_{\theta} = \{ (α \sim \tau) \mid [α \mapsto \tau] \in \theta \} \).

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 rigor 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($X$):

\[
\alpha \text{ fresh } \quad \Gamma, (x : \alpha) \mapsto e : \tau \sim C \\
\Gamma \mapsto \lambda x. e : (\alpha \rightarrow \tau) \sim C
\]

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 $W$ 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 $\{N, i \geq n\}$ are unique and unused) as state through the program, removing a name from the source when a fresh variable was introduced (i.e. incrementing $n$) [Narachewski & Nipkow, 1999]. While this approach is perhaps closer to how $W$ 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 $\exists \alpha$. $C$ ($\exists$ is used here rather than $\exists$ to distinguish between the two existential quantifiers in $C$; $\exists$ is for local assumptions, and $\exists$ is for unification variables). This approach is similar to the existential quantifiers used in [Pottier & Rémy, 2005](https://www.cl.cam.ac.uk/ftp/software/450/).

To use these quantifiers, we must first rearrange the constraint generation rules so that the type is viewed as
This means that the only place where these new unification variables are mentioned is within the generated metavariable input. The exact form of the inferred type is 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:

\[
\alpha \text{ fresh } \quad \beta \text{ fresh } \quad \Gamma, (x : \alpha) \vdash e : \beta \leadsto C
\]

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

\[
\alpha, \beta, \Delta, \Gamma, (x : \alpha) \vdash e : \beta \leadsto C
\]

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.

\footnote{This does not lead to ambiguity dangers, as the system is still syntax-directed}
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 ($\alpha$). 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.
Our main solving judgement is of the form $\langle Q; \theta \rangle \vdash C \rightarrow I$, which can be read as, “Given the axiom scheme $Q$ and constraint $Q_{\text{given}}$, the constraint $C_{\text{wanted}}$ is simplified to $Q_{\text{residual}}$ producing the substitution $\theta$ where $\text{dom}(\theta) \subseteq \alpha_{\text{ch}}$.” Note that while the solver is not obliged to solve all constraints, the remaining residual constraint is a $Q$-constraint, not a $C$-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!

### 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 $C$, 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 $C$ constraints, see fig. 5). Our solver operates on these separated constraints rather than the $C$-constraints directly, so we introduce a function $\text{sep} : C \rightarrow C$, defined in terms of helper functions $\text{simple} : C \rightarrow Q$ and $\text{implic} : C \rightarrow I$, which forms an additional phase in the inference algorithm.

In the original presentation of the $\text{OUTSIDEIN}(X)$ 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 $C$, $\text{simple}[C]$ and $\text{implic}[C]$ 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 troubles 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 form $\langle Q; \theta \rangle \vdash C_{\text{given}} \rightarrow C_{\text{wanted}} \rightarrow Q_{\text{residual}} \rightarrow \theta$, which can be read as, “Given the axiom scheme $Q$ and constraint $Q_{\text{given}}$, the constraint $C_{\text{wanted}}$ is simplified to $Q_{\text{residual}}$ producing the substitution $\theta$ where $\text{dom}(\theta) \subseteq \alpha_{\text{ch}}$.” Note that while the solver is not obliged to solve all constraints, the remaining residual constraint is a $Q$-constraint, not a $C$-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!
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 into 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 $\gamma$, use it for constraint generation, then apply the solution substitution $\theta$ to it in order to determine the type of $f$.
- As we now have an explicit notion of available type variables in the environment $\Delta$ 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 rigor 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:

```agda
record \( x \) : Set \( 1 \) where
  field Type : \ldots
  field QConstraint : \ldots
  \ldots

module OutsideIn (x : X) where
  open \( x \)
  \ldots
  data Constraint \ldots
  data Expression \ldots
```

(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 \( \text{Set}_1 \), rather than the usual type-of-types \( \text{Set} \), because the \( X \) parameter sits one meta-level higher than the definitions themselves — the type \( \text{Set} \) cannot contain \( \text{Set} \) without making Agda inconsistent\(^\text{10}\).

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 \( \alpha \)-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 \text{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 $\mathbb{N}$ to represent type variables:\footnote{Where $\mathbb{N}$ is the type of the standard Peano naturals with zero : $\mathbb{N}$ and suc : $\mathbb{N} \to \mathbb{N}$}

```plaintext
record X : Set₁ where  
  field Type : Set  
  Var : Π{N : 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  
  _\_ : Constraint → Constraint  
  _\_ : Π{N : 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 $\mathbb{N}$ 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\footnote{Morris \textit{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\footnote{McBride, 2003}. Reworking the above term definition to include such indexing, we get:

```plaintext
record X : Set₁ where  
  field Type : N → Set  
  Var : ∀{N : N} → FIN n → Type  
  →' : ∀{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  
  _\_ : Constraint (suc n) → Constraint n  
  _\_ : (m : N) → QConstraint n → Constraint (n + m) → Constraint n

Fig. 8: FIN-named constraint representation
```
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 \( \text{Fin} \ n \), where \( n \) is the number of available variables in the term. \( \text{Fin} \ n \) is a type containing exactly
\( n \) inhabitants — a finite set of natural numbers \([0, n]\) — defined as follows:

\[
\text{data} \quad \text{Fin} : \mathbb{N} \to \text{Set} \quad \text{where}
\begin{align*}
\text{zero} & : \forall n : \mathbb{N} \to \text{Fin} (\text{suc} n) \\
\text{suc} & : \forall n : \mathbb{N} \to \text{Fin} n \to \text{Fin} (\text{suc} n)
\end{align*}
\]

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 \( \forall. \forall. \text{QC}\ (\text{Var} \ \text{zero} \sim \text{Var} (\text{suc} \ \text{zero})) \) is closed and could
therefore be of type \( \text{Constraint} \ n \) 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, \( \text{QC} (\text{Var} \ \text{zero} \sim \text{Var} (\text{suc} \ \text{zero})) \), is by contrast
typed most generally as \( \text{Constraint} (\text{suc} \ (\text{suc} n)) \) 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:

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

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

\( \exists_._. \supset : (m : \mathbb{N}) \to \text{QConstraint} n \to \text{Constraint} (n + m) \to \text{Constraint} n \)

Note that the antecedent \( \text{QConstraint} \) has only \( n \) available variables, not \( n + m \), as it is impossible for the
antecedent 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 \( \text{Int} \) or \( \text{Maybe} \), 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 construc-
tors. 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 displeasing.

A common alternative is to introduce a separate \( \text{Con} \) introduction form for \( \text{Type} \), which refers to type construc-
tors, as opposed to the \( \text{Var} \) 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 \( \text{Con} \) 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 \text{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:

\[
\begin{aligned}
\text{record } X : \text{Set}_1 \text{ where} \\
\text{field } \text{Type} : & \text{Set} \to \text{Set} \\
\text{field } \text{Var} : & \forall \{n : \text{Set}\} \to n \to \text{Type} \\
\rightarrow' : & \forall \{n : \text{Set}\} \to \text{Type} n \to \text{Type} n \to \text{Type} n \\
\text{field } \text{QConstraint} : & \text{Set} \to \text{Set} \\
\epsilon : & \forall \{n : \text{Set}\} \to \text{QConstraint} n \\
\wedge : & \forall \{n : \text{Set}\} \to \text{QConstraint} n \to \text{QConstraint} n \to \text{QConstraint} n \\
\sim : & \forall \{n : \text{Set}\} \to \text{Type} n \to \text{Type} n \to \text{QConstraint} n
\end{aligned}
\]

\[
\begin{aligned}
\text{module OutsideIn (x : X) where} \\
\text{open X(x)} \\
\text{data CONSTRAINT (n : Set) : Set where} \\
QC : & \text{QConstraint} n \to \text{CONSTRAINT} n \\
\wedge' : & \text{CONSTRAINT} n \to \text{CONSTRAINT} n \to \text{CONSTRAINT} n \\
\vd : & \text{CONSTRAINT} (\mathcal{S} n) \to \text{CONSTRAINT} n \\
\exists_\_ \_ \ sibling_\_ : & (m : N) \to \text{QConstraint} n \to \text{CONSTRAINT} (n \oplus m) \to \text{CONSTRAINT} n
\end{aligned}
\]

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 \(\tau + \mathbb{1}\) and is therefore implemented as follows:

\[
\begin{aligned}
\text{data } \mathcal{S} (\tau : \text{Set}) : \text{Set} \text{ where} \\
\text{zero} : & \mathcal{S} \tau \\
\text{suc} : & \tau \to \mathcal{S} \tau
\end{aligned}
\]

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

\[
\begin{aligned}
\_ \oplus \_ : & \text{Set} \to N \to \text{Set} \\
x \oplus \text{zero} & = x \\
x \oplus \text{suc } n & = (\mathcal{S} x) \oplus n
\end{aligned}
\]

Then, a simplifier can take as arguments constraints of type \(\text{QConstraint} (x \oplus n)\), 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 \(\alpha\) 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 \(\text{unit} : \alpha \to m \alpha\) 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:

\[
\begin{aligned}
\text{Left Identity: } & \text{unit} \circ_m f = f \\
\text{Right Identity: } & g \circ_m \text{unit} = g \\
\text{Associativity: } & f \circ_m (g \circ_m h) = (f \circ_m g) \circ_m h
\end{aligned}
\]

\[\text{13 i.e. Haskell’s } \text{Maybe} \text{ type.}\]
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. 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 \( \alpha \to \text{Type} \beta \) as morphisms. Note that the type \( \alpha \to \text{Type} \beta \) is that of a substitution; Kleisli composition is substitution composition, where \( \text{Var} \) is the identity substitution. \( \text{Var} \) 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 \( \text{bind} \) function from Kleisli composition and \( \text{unit} \):

\[
\text{bind} : \forall \{\alpha \beta\} \to (\alpha \to \text{Type} \beta) \to \text{Type} \alpha \to \text{Type} \beta
\]

(Where “it” is a constructor for a unit type)

When applied to types, \( \text{bind} \) is clearly application of a substitution to a term:

\[
\text{bind} : \forall \{\alpha \beta\} \to (\alpha \to \text{Type} \beta) \to \text{Type} \alpha \to \text{Type} \beta
\]

Viewed categorically, \( \text{bind} \) could be viewed here as a mapping from morphisms in the Kleisli category of Type (i.e functions with types of the form \( \alpha \to \text{Type} \beta \)) 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 \( \text{Type} \tau \) for any \( \tau \). Note that the following two properties hold for \( \text{bind} \):

- **Identity:** \( \text{bind} \text{unit} = \text{id} \)
- **Composition:** \( \text{bind} \circ \text{bind} = \text{bind} \circ \text{bind} \)

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 \( \text{bind} \), we can say that \( \text{bind} \) 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:

\[
Q\text{-subst} : \forall \{\alpha \beta\} \to (\alpha \to \text{Type} \beta) \to (\text{QConstraint} \alpha \to \text{QConstraint} \beta)
\]

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

- **Identity:** \( Q\text{-subst} \text{unit} = \text{id} \)
- **Composition:** \( Q\text{-subst} \circ Q\text{-subst} = Q\text{-subst} \circ Q\text{-subst} \)

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 \( \ddot{\cdot} \cdot \cdot \cdot \) contains a subexpression of type CONSTRAINT \((S \alpha)\). Showing that \( S \) is a monad 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 \( \alpha \to \text{Type} \beta \) to a substitution \( S \alpha \to \text{Type} (S \beta) \), as required for this case. The form \( \exists n. Q \supset C \) contains a subexpression of type \( \text{CONSTRAINT} (\alpha \oplus n) \), which is a slightly more complicated beast to tame. Here, we must show not only that \( S \) is a monad, but that \( S \circ S \) is also, and \( S \circ S \circ S \), 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 \( \text{lift} \) from any monad \( M \) to \( M \circ S \). By showing that \( \text{lift} \) is monad homomorphism for the \( S \)-transformer, and that the resultant type \( M \circ S \) is a monad if \( M \) is a monad, it can be trivially shown that any number of \( S \)'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 \( \text{Types}, \text{QConstraints} \) and so on is renaming. Renaming can be implemented via functor composition in terms of substitution:

\[
\tau\text{-rename} : (\alpha \to \beta) \to (\text{Type} \alpha \to \text{Type} \beta)
\]

\[
\tau\text{-rename} f = \text{bind} (\text{rename } f)
\]

\[
Q\text{-rename} : (\alpha \to \beta) \to (\text{QConstraint} \alpha \to \text{QConstraint} \beta)
\]

\[
Q\text{-rename} f = \text{Q-subst} (\text{rename } f)
\]

Where \( \text{rename} \) is a functor from the base \( \text{Agda} \) category to the Kleisli category of \( \text{Type} \):

\[
\text{rename} : (\alpha \to \beta) \to (\alpha \to \text{Type} \beta)
\]

\[
\text{rename} f = \text{unit} \circ f
\]

With these definitions, the common de Bruijn index “up-shift” substitution which increments every index is trivially \( \text{rename } \text{suc} \). Up-shifting by multiple variables at a time can be done simply by \( \text{rename } \text{unit}_n \) where \( \text{unit}_n \) 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 \( \text{Agda} \)\(^{16} \) as a data type that reifies definitional equality:

\[
\text{data} \equiv \{A : \text{Set} \} : A \to A \to \text{Set} \ where
\]

\[
\text{refl} : \forall x : A \to x \equiv x
\]

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:

\[
\text{trans} : \forall \{A : \text{Set}\} \{x y z : A\} \to x \equiv y \to y \equiv z \to x \equiv z
\]

\[
\text{sym} \equiv \forall \{A : \text{Set}\} \{x y : A\} \to x \equiv y \to y \equiv x
\]

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:

\[
\textbf{postulate extensionality} : \{A \to B\} \to (\forall a \to f \circ a = g \circ a) \to f = g
\]

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 \( \text{refl} \)), 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 \( \text{error} \)), 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 \( \text{OutsideIn}(X) \) 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 \( \text{SeparatedConstraint} \), our representation of \( C \), 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:

\[
\text{sep} : \forall \{tv : \text{Set}\} \to \text{Constraint tv} \to \text{SeparatedConstraint tv}
\]

However our \( \text{sep} \) function is not total, as it is. The \( \text{sep} \) function defined in figure 5 is not defined for constraints of the form \( F \). To solve this problem, we adjust our constraint representation to be indexed by a \( \text{strata} \):

\[
\text{module OutsideIn } (x : X) \text{ where}
\]

\[
\begin{align*}
\text{record } X : \text{Set}_1 \text{ where} & \quad \text{field } \text{Type} : \text{Set} \to \text{Set} \\
& \quad \text{Var} : \forall n : \text{Set} \to n \to \text{Type } n \\
& \quad \to' : \forall n : \text{Set} \to \text{Type } n \to \text{Type } n \to \text{Type } n \\
\text{field } \text{QConstraint} : \text{Set} \to \text{Set} & \quad \text{field } \epsilon : \forall n : \text{Set} \to \text{QConstraint } n \\
& \quad \land : \forall n : \text{Set} \to \text{QConstraint } n \to \text{QConstraint } n \to \text{QConstraint } n \\
& \quad \sim : \forall n : \text{Set} \to \text{Type } n \to \text{Type } n \to \text{QConstraint } n \\
\text{module } OutsideIn \text{ where} & \quad \text{open } X(x) \\
\text{data } \text{Strata} : \text{Set} \text{ where} & \quad \text{Hi} : \text{Strata} \\
& \quad \text{Lo} : \text{Strata} \\
\text{data } \text{Constraint} \ (n : \text{Set}) : \text{Strata} \to \text{Set} \text{ where} & \quad \text{QC} : \forall \sigma : \text{Strata} \to \text{QConstraint } n \to \text{Constraint } n \sigma \\
& \quad \land' : \forall \sigma : \text{Strata} \to \text{Constraint } n \sigma \to \text{Constraint } n \sigma \to \text{Constraint } n \sigma \\
& \quad \lor : \text{Constraint } (\sigma \land' n \sigma) \text{Hi} \to \text{Constraint } n \text{Hi} \\
& \quad \exists_\_ \_ \supset_\_ \_ : \forall \sigma \to (m : \mathbb{N}) \to \text{QConstraint } n \to \text{Constraint } (n \supset m) \sigma \to \text{Constraint } n \sigma
\end{align*}
\]

Fig. 10: Final Constraint Representation

\(^{17}\) A \( \text{setoid} \) being a dependent product \((\tau, \sim)\) where \( \sim \) is an equivalence relation on the type \( \tau \).
Fig. 11: A structurally recursive prenexer

Note that the type \( \text{Constraint} \ n \ Hi \) contains all forms of constraints, whereas the type \( \text{Constraint} \ n \ Lo \) 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 \( \text{sep} \) function, so as to make it total:

\[
\text{sep} : \forall \{tv : \text{Set}\} \to \text{Constraint} \ tv \ Lo \to \text{SeparatedConstraint} \ tv
\]

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

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

Implementing the prenexer as presented in figure 10 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 \( \text{Expression} \)s 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:

\[
\text{data} \ \text{Expression} \ (ev \ tv : \text{Set}) : \cdots
\]

This approach brings a number of advantages. For example, type environments like \( \Gamma \) 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 \( \text{type} \) (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 \( \text{Name} ev \), defined as follows:

\[
\text{data} \ \text{NameType} : \text{Set} \ \text{where}
\begin{align*}
\text{Binding} & : \ \text{NameType} \\
\text{Datacon} & : \ N \to \text{NameType}
\end{align*}
\]

\[
\text{data} \ \text{Name} (n : \text{Set}) : \ \text{NameType} \to \text{Set} \ \text{where}
\begin{align*}
N & : n \to \text{Name} n \ \text{Binding} \\
DC & : \forall \{x\} \to \text{dc} \ x \to \text{Name} n \ (\text{Datacon} \ x)
\end{align*}
\]

(Where \( \text{dc} \) is a type for datacon names, indexed by their arity, provided in the parameter \( X \))
arity is presented in the type Name, we can easily produce the desired type for a pattern matching alternative:

\[
\textbf{data Alternative} \ (ev\ tv : \ Set) \ : \ \text{SHAPE} \to \text{Set} \ \textbf{where} \\
\text{alt} \ : \ \forall \{n : \mathbb{N}\} \to \text{Name} \ ev \ (\text{Datacon} \ n) \to \text{Expression} \ (ev \ \oplus \ n) \ tv \to \\
\text{Alternative} \ ev \ tv
\]

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 \(\text{VarCon}\) and \(\text{CASE}\):

\[
\frac{\alpha, \beta, \gamma, \delta, \Delta; \Gamma \vdash e : \alpha \to C}{\Delta; \Gamma \vdash v : \tau \to \Delta \alpha. [\Delta \to \alpha]Q_1 \wedge' (\tau \sim [\Delta \to \alpha]\tau_1)} \ \
\text{VarCon}
\]

\[
\frac{(K_i : \forall \overline{a}. Q_i \Rightarrow \overline{\tau_i} \to T \overline{\alpha}) \in \Gamma}{\alpha, \beta, \gamma, \delta, \Delta; \Gamma \vdash e : \alpha \to C}
\]

\[
\frac{C_i' = \begin{cases} C_i \wedge' \delta_i \sim \beta & \text{if} \ \overline{b_i} = e \ \text{and} \ \overline{Q_i} = e \\ \pi \overline{\beta}, \overline{\delta}. (\overline{[a \mapsto \rho]}[\Delta \to \gamma]Q_1) \supseteq C_i \wedge' \delta_i \sim \beta & \text{otherwise} \end{cases}}{\Delta; \Gamma \vdash \text{case} e \ \text{of} \ \{K_i, x_i \to e_i\} : \tau \sim \exists \overline{\beta}, \overline{\delta}. C \wedge' (T \overline{\gamma} \sim \alpha) \wedge' (\bigwedge C_i') \wedge' (\tau \sim \beta)} \ \
\text{CASE}
\]

Note that in each instance, the form of the type schema retrieved from the environment is different! One form for data constructors (\(\forall \overline{a}. Q \Rightarrow \overline{\tau} \to T \overline{\alpha}\)), and another for bindings (\(\forall a. Q \Rightarrow \tau\)). As \(\text{Type}\) 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 \(\text{NameType}\), providing a general type schema form for types of regular \(\text{Bindings}\) but more structured forms for types of data constructors:

\[
\textbf{data TypeSchema} \ (tv : \ Set) \ : \ \text{NameType} \to \text{Set} \ \textbf{where} \\
\forall'_\ldots \Rightarrow \ldots : \ \forall' \{n : \mathbb{N}\} \to \text{QConstraint} \ (tv \ \oplus \ n) \to \text{Type} \ (tv \ \oplus \ n) \to \text{TypeSchema} \ tv \ \text{Regular} \\
\forall' \ldots \Rightarrow \ldots : \ \forall' \{a : \mathbb{N}\} \{r : \mathbb{N}\} \to \text{VEC} \ (\text{Type} \ (tv \ \oplus \ a)) r \to tv \to \text{TypeSchema} \ tv \ (\text{Datacon} \ r) \\
\forall' \ldots \Rightarrow \ldots : \ \forall' \{a b : \mathbb{N}\} \{r : \mathbb{N}\} \to \text{QConstraint} \ (tv \ \oplus \ a \ \oplus \ b) \to \\
\text{VEC} \ (\text{Type} \ (tv \ \oplus \ a \ \oplus \ b)) r \to tv \to \text{TypeSchema} \ tv \ (\text{Datacon} \ r)
\]

Here, we use a length-indexed vector \(\text{VEC}\) to ensure that the arity of the function type for data constructors matches the arity mentioned in their \(\text{Name}\).

This allows us to define environments as a total mapping of any \(\text{Name}\) of type \(n\) to \(\text{TypeSchema}\) for the same name type \(n\).

\[
\begin{align*}
\text{Environment} & : \text{Set} \to \text{Set} \\
\text{Environment} \ ev\ tv &= \forall \{n : \text{NameType}\} \to \text{Name} \ ev \ n \to \text{TypeSchema} \ tv \ n
\end{align*}
\]

### 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:

\[
\frac{\alpha, \beta, \Delta; \Gamma, (x : \alpha) \vdash e : \beta \to C}{\Delta; \Gamma \vdash \lambda x. e : \tau \to \exists \overline{\alpha}. \beta. C \wedge' (\tau \sim (\alpha \to \beta))} \ \text{Abs}
\]

In this rule, two new type variables are introduced into the set of available variables \(\Delta\). This corresponds in our representation to an index \(S\ (S\ tv)\), however the expression \(e\)'s index is merely \(tv\). The obvious way to remedy this is to simply "up-shift" the de Bruijn indices in \(e\) twice using the appropriate substitution functor,
data NameType : Set where
  Binding : NameType
  Datacon : ℕ → NameType

data Name (n : Set) : NameType → Set where
  N : n → Name n Binding
  DC : ∀ {x} → dc x → Name n (Datacon x)

data Alternative (ev tv : Set) : Shape → Set where
\texttt{alt} : ∀{n : N}\{σ : Shape\} → Name ev (Datacon n) → Expression (ev ⊕ n) tv σ →
  Alternative ev tv (Unary σ)

data Alternatives (ev tv : Set) : Shape → Set where
esac : Alternatives ev tv Nullary
| : ∀[σ₁ σ₂ : Shape] → Alternative ev tv σ₁ → Alternatives ev tv σ₂ →
  Alternatives ev tv (Binary σ₁ σ₂)

data Expression (ev tv : Set) : Shape → Set where
EVar : ∀{x : NameType} → Name ev x → Expression ev tv Nullary
X' : ∀[σ : Shape] → Expression (S ev) tv σ → Expression ev tv (Unary σ)
app : ∀[σ₁ σ₂ : Shape] → Expression ev tv σ₁ → Expression ev tv σ₂ →
  Expression ev tv (Binary σ₁ σ₂)
let : ∀[σ₁ σ₂ : Shape] → Expression ev tv σ₁ → Expression (S ev) tv σ₂ →
  Expression ev tv (Binary σ₁ σ₂)
let₁ : ∀[σ₁ σ₂ : Shape] → Expression ev tv σ₁ → Type tv →
  Expression (S ev) tv σ₂ →
  Expression ev tv (Binary σ₁ σ₂)
let₁ₙ : ∀[σ₁ σ₂ : Shape] → (n : ℕ) → Expression ev (tv ⊕ n) σ₁ →
  QConstraint (tv ⊕ n) → Type (tv ⊕ n) →
  Expression (S ev) tv σ₂ →
  Expression ev tv (Binary σ₁ σ₂)
case : ∀[σ₁ σ₂] → Expression ev tv σ₁ →
  Alternatives ev tv σ₂ →
  Expression ev tv (Binary σ₁ σ₂)

Fig. 12: The complete Expression representation

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 \(e\)-subst \((\text{rename } (\text{suc } \circ \text{suc})) \ e\). 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 \textit{third} index to the \(expression\) type, which represents the structure or
the \textit{shape} of the expression tree (see fig. [12]). The \textit{Shape} type itself is a straightforward tree structure which is
sufficient to encode the recursion pattern of constraint generation:

data Shape : Set where
  Nullary : Shape
  Unary : Shape → Shape
  Binary : Shape → Shape → Shape

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

\(e\)-subst : ∀[σ : Shape] \{ev α β : Set\} → (α → Type β) → Expression ev α σ → Expression ev β σ

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 representa-
tion for the \textit{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):

\[
\text{gen\text{-}Constraint : } \{ \text{ev tv : Set} \} \{ \sigma : \text{Shape} \} \\
\quad \rightarrow \text{Environment ev tv} \\
\quad \rightarrow \text{Expression ev tv } \sigma \\
\quad \rightarrow \text{Type tv} \\
\quad \rightarrow \text{Constraint tv}
\]

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:

\[
\text{syntax ConstraintGen } \Gamma \tau e C = \Gamma \mapsto e : \tau \mapsto C \\
\text{data ConstraintGen } \{ \text{ev tv : Set} \} \{ \Gamma : \text{Environment ev tv} \} \{ \tau : \text{Type tv} \} : \\
\quad \{ \sigma : \text{Shape} \} \rightarrow \text{Expression ev tv } \sigma \rightarrow \text{Constraint tv Extended} \rightarrow \text{Set} \text{ where}
\]

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:

\[
\text{gen\text{-}Constraint} : \{ \text{ev tv : Set} \} \{ \sigma : \text{Shape} \} \{ \Gamma : \text{Environment ev tv} \} \{ \tau : \text{Type tv} \} \\
\quad \rightarrow \exists \lambda C \rightarrow \Gamma \mapsto e : \tau \mapsto C
\]

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:

\[
\begin{align*}
\text{Q}; Q_g; \bar{\alpha} &\mapsto \text{Q} \mapsto Q_r; \theta \\
\forall (\exists [I, \bar{\beta}, Q_s \supset C_i] \in I). \text{Q}; Q_g \land Q_r \land Q_i; \bar{\beta} &\mapsto \text{C}_i \mapsto e; \theta_i \\
\text{Q}; Q_g; \bar{\alpha} &\mapsto \text{Q} \cdot I \mapsto Q_r; \theta
\end{align*}
\]

In particular, note that each implication constraint body must resolve to \(\epsilon\). 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 \(\epsilon\). As \(\text{QConstraints}\) 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 \(\epsilon\) in the \(X\) constraint:

\[
\text{field dec-}\epsilon : (x : \text{QConstraint}) \rightarrow \text{Dec} (x \equiv \epsilon)
\]

Interestingly, as this function must be total and terminating, this can be viewed as a proof of the decidability of the \text{Constraint-Gen} proposition.
Where Dec is defined (in the standard library) as follows:

\[
\text{data Dec } (P : \text{Set}) : \text{Set where}
\]
\[
\begin{align*}
\text{yes} & : P \rightarrow \text{Dec } P \\
\text{no} & : (P \rightarrow \bot) \rightarrow \text{Dec } P
\end{align*}
\]

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 $\text{OUTSIDEIN}(X)$ 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 instantiatable.\footnote{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!}

Our instantiation of $\text{QConstraint}$ is the smallest that meets the requirements of the $X$ parameter. It consists only of $\epsilon$, constraint conjunction and type equality constraints:

```haskell
data QConstraint (x : Set) : Set where
  ~ : Type x → Type x → QConstraint x
  ∧ : QConstraint x → QConstraint x → QConstraint x
  ϵ : QConstraint x
```

Types consist of type variables (or constructors), type application and function types — the typical assortment of types from System F.

```haskell
data Type (x : Set) : Set where
  Var : n → Type n
  →′ : Type n → Type n → Type n
  app : Type n → Type n → Type n
```

Types are 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:

```haskell
Q : Q_g ; ω_tch ; Q_w ; unitType
```

While this simplifier meets the simulator 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 $\theta$ given two terms $\tau_1$ and $\tau_2$ such that $\theta \tau_1 = \theta \tau_2$. 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 Type terms do not come indexed by the number of available variables they contain. Rather, they come in the form $\text{Type} (\tau \oplus n)$, with skolem variables and constructor names represented by the type $\tau$, and $n$ unification variables available. We define our unification algorithm in terms of a special
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:

\[
\begin{align*}
\text{data } & \text{ SNAME } (sk : \text{ Set }) (an : N) : \text{ Set where} \\
\text{ unification } & : \text{ Fin } un \rightarrow \text{ SNAME } sk \ an \\
\text{ rigid } & : \text{ sk } \rightarrow \text{ SNAME } sk \ an
\end{align*}
\]

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

\[
\begin{align*}
\text{iso}_1 : \forall \{m : N\} \{t : \text{ Set}\} \rightarrow t \oplus m \rightarrow \text{ SNAME } t \ m \\
\text{iso}_2 : \forall \{m : N\} \{t : \text{ Set}\} \rightarrow \text{ SNAME } t \ m \rightarrow t \oplus m
\end{align*}
\]

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 $\alpha_{\text{tech}}$, we omit mention of $\alpha_{\text{tech}}$ in figure 13.

### 4.2 Solving

The overall simplifier has to deal with the three possible constraint forms (see fig. 13). Empty $\epsilon$ 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 for an identical constraint to the wanted equality...
within the given context in order to trivially resolve the wanted constraint; and \texttt{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: \texttt{Mgu} is attempted first, then \texttt{Entail}, then \texttt{GiveUp}.

Another problem presents itself when one attempts to encode this simplifier in Agda. In the rule \texttt{Conj}, the first substitution $\theta$ is applied to the second conjunct $Q_2$, and recursion occurs on the substituted constraint $\theta Q_2$. This recursion is \textit{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 \texttt{Expression}s previously. We shall define a new form of constraint, indexed by its \textit{shape}:

\begin{verbatim}
data SConstraint (x : Set) : Shape → Set where
  ~   : TYPE x → TYPE x → SConstraint x Nullary
  ∧  : ∀{s₁ s₂} → SConstraint x r₁ → SConstraint x r₂ → SConstraint x (Binary s₁ s₂)
  ε  : SConstraint x Nullary
\end{verbatim}

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

$$SC\text{-sub} : ∀[\alpha \beta] \{s\} → (\alpha → TYPE \beta) → SConstraint \alpha s → SConstraint \beta s$$

As with \texttt{Expression}s, we can now glean from the type that the \textit{shape} of the constraint term does not change with substitution. Therefore, structural recursion can occur on the \textit{Shape} index of the \texttt{SConstraint} term, rather than on the term itself. As \texttt{SConstraint} and \texttt{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 \texttt{let}-generalisation:

\begin{verbatim}
data IntOrBool :: * → * where
  IsInt    :: (τ ~ Int) ⇒ IntOrBool τ
  IsBool   :: (τ ~ Bool) ⇒ IntOrBool τ

f :: IntOrBool α → α → Bool
f x y = let g z = not y in
  case x of
    IsInt → True
    IsBool → g ()
\end{verbatim}

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

\begin{verbatim}
f :: IntOrBool α → α → Bool
f x y = let g : ∀β. (α ~ Bool) → β → Bool
g z = not y in
  case x of
    IsInt → True
    IsBool → g ()
\end{verbatim}

The program now type-checks in our simple instantiation of \texttt{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 $\tau$ is inferred for an expression $e$, then there exists a proof according to some natural, permissive typing rules that $e$ is well typed by $\tau$. 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 $\epsilon$ 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 $\epsilon$. 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

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.

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


