CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A New Synthesis Tool for Agda

Master's thesis in Computer science and engineering

Lukas Skystedt

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

A New Synthesis Tool for Agda

LUKAS SKYSTEDT

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A New Synthesis Tool for Agda
LUKAS SKYSTEDT

Supervisor: Ulf Norell, Department of Computer Science and Engineering
Examiner: Moa Johansson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

A New Synthesis Tool for Agda

Lukas Skystedt
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Agda is a dependently typed functional programming language and proof assistant based on constructive type theory. Agda programmers spend most of their time working with unfinished programs where uncompleted parts are marked by *holes*. These holes can stand for anything from the core, complex parts of the program to trivial details. To allow programmers to focus on the complex parts of the program, Agda includes a synthesis tool called Agsy that can automatically fill in simple holes. Alas, the development of Agsy has not kept up with the evolution of Agda. Hence, Agsy fails to fill some trivial holes. This report describes a replacement for Agsy, named Mimer. To explain the methods used by Mimer, it also presents synthesis algorithms for two simple languages with meta-variables. The first is a simply typed λ-calculus with products and sums. The second is a dependently typed λ-calculus.

Acknowledgements

I would like to thank Ulf Norell for his supervision and insightful feedback.

Lukas Skystedt, Gothenburg, July 2022

Contents

1
Introduction

Agda [17] is a dependently typed functional programming language. It is based on constructive type theory which has a logical interpretation via the Curry-Howard correspondence. In the correspondence, types represent logical propositions. A proposition is proven by giving a term –a program– of the type representing it. When there are no possible programs of a type it is said to be empty and represents a proposition that is not provable. Type checking can be thought of as checking proof correctness. This way, Agda is also a *proof assistant*. Agda programmers typically switch between thinking of terms as programs and as proofs within the same project. They intend some parts to be executed (e.g. a sorting function) and some to only be type checked (e.g. a proof that the sorting function is correct).

Agda programmers can write partial terms in Agda by putting *holes* in place of missing parts. In practice, some holes stand for very complex terms while others stand for trivial details. Simple holes can be filled automatically by program *synthesizes*. The Curry-Howard correspondence naturally extends to the synthesis problem — program synthesis corresponds to proof automation. There is, however, a distinction between the two. For proofs, programmers care that *some* term is synthesized and are mostly indifferent to that term's *behavior* as a program. For programs, however, the behavior is their raison d'être. For example, a programmer might be indifferent to which term is used to prove that addition is commutative, but care deeply whether a list concatenation function actually produces the correct result. This report primarily takes the view of terms as proofs.

Agda's compiler includes a set of commands to interact with a program, one of which is a synthesis tool called Agsy [13, 12] that can fill simple holes. The implementation of Agsy does not work with Agda terms directly. Instead, it translates them to its own language, performs synthesis, and translates the result back to Agda. A drawback of this design is that discrepancies between the two languages easily arise. Since Agda is a research language, it changes frequently which exasperates the issue. Indeed, Agsy fails in some cases where it ought not to and completely breaks in the presence of co-patterns — one of the more recent additions to Agda.

These problems are the motivation for Mimer, a new synthesis tool intended to replace Agsy. It has been implemented as a component in Agda's compiler. In contrast to Agsy, Mimer works directly with Agda terms. It uses a type-directed search algorithm based on iterative refinement of meta-variables (holes). Mimer has been compared to Agsy on 77 example problems, showing that it supports a greater

range of language features but is slower. Also, in contrast to Agsy, Mimer does not (yet) perform case splitting or equality reasoning.

Agda is a complex language and it does not have a formal specification. Hence, a direct exposition of Mimer would be somewhat clumsy. For this reason, and to make the report more accessible to readers who are not familiar with dependent types, the Curry-Howard correspondence, or proof search, this report describes synthesis in three increasingly complex languages. First, chapter 2 describes a simply typed λ-calculus, $\lambda^{?}_{\rightarrow}$, gives some examples of the Curry-Howard correspondence in practice, and gives an algorithm for type-driven synthesis. Second, chapter 3 extends the ideas to a dependently typed language, $\lambda^{?}_{\Pi}$, based on a tutorial implementation of dependent types [14]. Third, chapter 4 presents the new synthesizer for Agda, Mimer.

2

Simply Typed λ-Calculus ($\lambda^{?}_{\to}$)

This chapter presents $\lambda^{?}_{\to}$ — a simply typed λ-calculus with products, sums, and meta-variables. It starts with an introduction to the language (section 2.1) and its relation to intuitionistic propositional logic (section 2.2). Then, section 2.3 aims to build up the reader's intuition for a synthesis algorithm. Finally, section 2.4 gives a complete synthesis algorithm and describes some of its properties. A full implementation is available online[1].

2.1 The Language $\lambda^{?}_{\to}$

The language's syntax is given in figure 2.1. There are three sorts of language constructions: program terms, types, and contexts (lists of bound variables and their types). For simplicity, I adopt Barendregt's variable convention — all variable names are assumed to be unique, and I have left out freshness conditions from the language rules. The implementation, like Agda, uses De Bruijn indices for bound variables. With the exception of meta-variables, the calculus is mostly standard and thorough treatments –including type checking algorithms– can be found elsewhere (e.g. [14, 18]). Meta-variables are the mechanism that $\lambda^{?}_{\to}$ uses for representing holes –missing parts of program terms– and are central to the synthesis algorithm. The meta-variables, unlike program variables, are not bound and used when evaluating programs but are used to reason *about* programs.

[1] `https://github.com/Lukas-Skystedt/stlc-synthesis`

Types	A, B	$::=$	$\mathcal{B}$	base types
		$\mid$	$A \to B$	function space
		$\mid$	$A \times B$	product type
		$\mid$	$A \uplus B$	sum type
		$\mid$	$\bot$	bottom (empty type)
Terms	s, t, u	$::=$	x	variables
		$\mid$	$\lambda x.t$	λ-abstraction
		$\mid$	$s\ t$	application
		$\mid$	(s, t)	pair
		$\mid$	$\texttt{fst}\ t$	first projection
		$\mid$	$\texttt{snd}\ t$	second projection
		$\mid$	$\uplus\texttt{-elim}\ s\ t\ u$	elimination of sum
		$\mid$	$\texttt{inj}_1\ t$	first injection
		$\mid$	$\texttt{inj}_2\ t$	second injection
		$\mid$	$\bot\texttt{-elim}\ t$	elimination of $\bot$
		$\mid$	$t :: A$	type annotation
		$\mid$	$?_n$	meta-variable
Contexts	Γ	$::=$	ε	empty context
		$\mid$	$\Gamma, x : A$	extending context

Figure 2.1: Syntax for $\lambda^{?}_{\to}$ types, terms, and contexts. Definition is for base types $\mathcal{B}$, variables x, and natural numbers n.

The typing rules are given in figure 2.2 and use the standard judgment form $\Gamma \vdash t : A$ which states that t has type A in context Γ. The implementation of $\lambda^{?}_{\to}$ uses a bidirectional type checking algorithm where type inference and type checking are interleaved. When *checking* the type of a meta-variable, the type checker simply notes what type it is supposed to have so that information is available later for the synthesizer. However, since a meta-variable can have any type, trying to *infer* the type of a meta-variable simply raises an error[2].

For synthesis, it will be useful to divide the rules into introduction rules and elimination rules. Introduction rules (Lam, Pair, Inj_1, Inj_2, Var, Meta) constructs types using type formers such as $\times$ while elimination rules (Fst, Snd, $\uplus$E, $\bot$E, App) deconstructs them.

[2] A similar problem of types not being unique occurs for sum types. Assume x : A. The type of $\texttt{inj}_1\ x$ must be on the form A $\uplus\ B$, but B cannot be inferred.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\textsc{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\textsc{Lam} \qquad \frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B}\textsc{App}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash (s,t) : A \times B}\textsc{Pair} \qquad \frac{\Gamma \vdash (s,t) : A \times B}{\Gamma \vdash s : A}\textsc{Fst} \qquad \frac{\Gamma \vdash (s,t) : A \times B}{\Gamma \vdash t : B}\textsc{Snd}$$

$$\frac{\Gamma \vdash t : \bot}{\Gamma \vdash \bot\texttt{-elim}\ t : A}\bot\textsc{E} \qquad \frac{\Gamma \vdash s : A \uplus B \qquad \Gamma \vdash t : A \to C \qquad \Gamma \vdash u : B \to C}{\Gamma \vdash \uplus\texttt{-elim}\ s\ t\ u : C}\uplus\textsc{E}$$

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash \texttt{inj}_1\ s : A \uplus B}\textsc{Inj}_1 \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \texttt{inj}_2\ t : A \uplus B}\textsc{Inj}_2 \qquad \frac{}{\Gamma \vdash ?_n : A}\textsc{Meta}$$

Figure 2.2: Typing rules for $\lambda^{?}_{\to}$

2.2 $\lambda^{?}_{\to}$ for Proofs

This section gives an informal treatment of the correspondence between simply typed λ-calculus and intuitionistic propositional logic. For a formal account, refer to, e.g., [8]. By the Curry-Howard correspondence, types are interpreted as propositions in the following manner: base types ($\mathcal{B}$) correspond to propositional atoms; the empty type ($\bot$) corresponds to a false proposition; function spaces ($\to$) to implication ($\Rightarrow$), product types ($\times$) to conjunction ($\wedge$), and sum types ($\uplus$) to disjunction ($\vee$). Note that no type constructor corresponds to negation ($\neg$). Instead, negation is taken to be implication to the false proposition[3]: $\neg A := A \to \bot$.

The typing judgment, $t : A$, can either be thought of as stating that t is a program term of type A or that t is a proof of the proposition A. Similarly, $\Gamma \vdash t : A$, stating that t has type A in context Γ, can be understood as t being a proof of A given a set of assumptions, Γ. The typing rules correspond to rules of logical inference in natural deduction. It can easily be seen by syntactically "erasing" the terms from the rules and making the contexts implicit. For example, the App rule corresponds to modus ponens ($\Rightarrow$-elimination):

$$\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B} \quad \mapsto \quad \frac{A \Rightarrow B \qquad A}{B}$$

For example, consider the tautology $(A \Rightarrow B) \wedge (A \Rightarrow C) \Rightarrow (A \Rightarrow B \wedge C)$. It can be proven in $\lambda^{?}_{\to}$ as follows:

$$\lambda p.\lambda a.(\texttt{fst}\ p\ a, \texttt{snd}\ p\ a) : (A \to B) \times (A \to C) \to (A \to B \times C)$$

If the term is thought of as a program, it is the higher-order function that takes two functions ($A \to B$ and $A \to C$) as arguments and combines them into a new function.

[3] Negation is often defined this way in intuitionistic logic as well.

2.3 Proof Synthesis

A synthesis algorithm should take a meta-variable with a goal type as input and produce a term of that type that is free from meta-variables. The underlying idea that makes this possible is a particular reading of the typing rules. For the given goal type, one can identify all the typing rules that can have it in the conclusion. Then, the goal can be speculatively *refined* using each applicable rule in turn — reducing the synthesis problem to the synthesis of the assumptions of the rule, which are solved recursively. I will now illustrate this approach using a simple example; the synthesis of one of De Morgan's Laws (specialized to the base types A, B, C):

$?_1$ where
$\varepsilon \vdash ?_1 : \neg A \uplus \neg B \to \neg(A \times B)$

The goal type has a function arrow as the top-level type former. That type matches all the rules except $\textsc{Inj}_1$ and $\textsc{Inj}_2$ (they produce types on the form $X \uplus Y$) and Pair (it produces types on the form $X \times Y$). However, we can drastically reduce the search space by utilizing η-equality: If $t : X \to Y$, then $\lambda x.t\ x : X \to Y$. The equality entails that we may restrict the search to terms on η-long form by using just the Lam rule when the goal type is an arrow. In our example, refining using the Lam rule yields:

$\lambda s.\ ?_2$ where
$s : \neg A \uplus \neg B \vdash ?_2 : \neg(A \times B)$

Recall that $\neg X := X \to \bot$, so the goal is of arrow type again. We refine using Lam again[4]:

$\lambda s\ p.\ ?_3$ where
$s : \neg A \uplus \neg B,\ p : A \times B \vdash ?_3 : \bot$

Now the goal type is $\bot$, so the Lam rule is not applicable. Like before, the introduction rules for sums and products are not applicable either. We may then try the variable rule, but the context does not contain a variable of type $\bot$. Next up is the App rule. However, because it is *always* applicable, using it blindly gives an ill-behaved non-terminating algorithm. We will remedy this later by integrating application with variable usage (Var). Next, we could try the rule ⊎E. If we try to use it directly to produce the term $\lambda s\ p.\ \uplus\texttt{-elim}\ ?_4\ ?_5\ ?_6$ we run into an issue: What are the goal types of the new meta-variables? We can infer that $?_5 : \alpha \to \bot$ and $?_6 : \beta \to \bot$ which entails $?_4 : \alpha \uplus \beta$, but we cannot decide what α and β should be. We can remedy the situation using two identities for sums:

$$\uplus\texttt{-elim}\ (\texttt{Inj}_1\ t)\ f\ g \equiv f\ t$$
$$\uplus\texttt{-elim}\ (\texttt{Inj}_2\ t)\ f\ g \equiv g\ t$$

[4] I write $\lambda a\ b.t$ as syntactic sugar for $\lambda a.\lambda b.t$.

These, in combination with $\beta\eta$-equality, entail that the only situation where we need $\uplus$E is when a sum is derived from the context with application and projections. In our example, we can use a sum from the context directly:

$$\begin{aligned}&\lambda s\ p.\ \uplus\texttt{-elim}\ s\ ?_4\ ?_5 \quad \text{where}\\ &\quad s : \neg A \uplus \neg B,\ p : A \times B\ \vdash ?_4 : \neg A \to \bot\\ &\quad s : \neg A \uplus \neg B,\ p : A \times B\ \vdash ?_5 : \neg B \to \bot\end{aligned}$$

For the first time, we get two sub-goals. For consistency with later algorithms, we will solve them right-to-left even though it does not matter for $\lambda^{?}_{\to}$. The goal $?_5$ has an arrow type, so we λ-abstract:

$$\begin{aligned}&\lambda s\ p.\ \uplus\texttt{-elim}\ s\ ?_4\ (\lambda b.?_6) \quad \text{where}\\ &\quad s : \neg A \uplus \neg B,\ p : A \times B\ \vdash ?_4 : \neg A \to \bot\\ &\quad s : \neg A \uplus \neg B,\ p : A \times B, b : \neg B\ \vdash ?_6 : \bot\end{aligned}$$

The goal is now $\bot$. We have nothing of exactly that type in the context, but $b : B \to \bot$ has the right type when applied to a term of type B. Here, the integration of the Var rule and the App is useful — we simply refine the goal with b applied to a new meta-variable.

$$\begin{aligned}&\lambda s\ p.\ \uplus\texttt{-elim}\ s\ ?_4\ (\lambda b.b\ ?_7) \quad \text{where}\\ &\quad s : \neg A \uplus \neg B,\ p : A \times B\ \vdash ?_4 : \neg A \to \bot\\ &\quad s : \neg A \uplus \neg B,\ p : A \times B, b : \neg B\ \vdash ?_7 : B\end{aligned}$$

We now need something of type B and we have $p : A \times B$ in the context. Like with $\uplus$E, we have identities that justify only applying Fst and Snd directly to terms derived from the context:

$$\begin{aligned}\texttt{fst}\ (s, t) &\equiv s\\ \texttt{snd}\ (s, t) &\equiv t\end{aligned}$$

The integrated version of the Snd rule gives us:

$$\begin{aligned}&\lambda s\ p.\ \uplus\texttt{-elim}\ s\ ?_4\ (\lambda b.b\ (\texttt{snd}\ p)) \quad \text{where}\\ &\quad s : \neg A \uplus \neg B,\ p : A \times B\ \vdash ?_4 : \neg A \to \bot\end{aligned}$$

The solution to $?_4$ is entirely analogous to the solution to $?_5$. The final term is

$$\lambda s\ p.\ \uplus\texttt{-elim}\ s\ (\lambda a.a\ (\texttt{fst}\ p))\ (\lambda b.b\ (\texttt{snd}\ p)) : \neg A \uplus \neg B \to \neg(A \times B)$$

In summary, we can create a synthesis algorithm by iteratively refining goals using the typing rules. However, one should be careful about how and when to apply the rules to avoid loops and generating sub-goals with unknown types.

2.4 Synthesis Algorithm

Based on the observations in the previous section, we can now give a full algorithm for term synthesis. We saw that there are two main problems with blindly refining goals with *any* matching typing rules. First, speculatively applying an elimination rule –such as Fst– yields a new goal that can be filled by the corresponding introduction rule (Pair in the case of Fst). Thus, the algorithm would, at every step, generate branches that will be refined by elimination-introduction pairs indefinitely. Second, a goal's type does not uniquely determine the types of the sub-goals when refining by $\uplus$E.

We solve both issues by applying elimination rules only to variables in the context – possibly multiple times. Hence, the algorithm consists of two parts. The first part generates possible components from the context by iteratively applying eliminations rules: Var, Fst, Snd, $\uplus$E, $\bot$E. The other part refines goals using components or introduction rules: Lam, Pair, Inj_1, Inj_2. Figure 2.3 shows rules for the algorithm. I use the following notation:

$\Gamma \mid ? : A$	Goal: Find a term of type A in context Γ
ε	Empty list (of goals)
$g :: gs$	Cons: List with head g and tail gs
$gs +\!\!+ gs'$	Concatenate lists
σ	Substitution of meta-variables for terms
$[?_i \mapsto t]\sigma$	Extending a substitution
$\sigma; gs$	A search branch is a substitution and a list of goals

$\texttt{AppFull}_\Gamma$ applies a term to as many new meta-variables as possible, returning the new term and the new sub-goals. For example,

$$\texttt{AppFull}_\Gamma(f : A \to B \to C) = (f\ ?_1\ ?_2,\ \{\Gamma \mid ?_1 : A,\ \Gamma \mid ?_2 : B\}).$$

Recall that because we always generate terms on η-long form, it is safe to restrict the set of components to the fully applied ones. The set of components $\mathcal{C}_\Gamma$ is the largest set generated by the component rules for a specific context, Γ. The refinement rules are written on a form to be reminiscent of the typing rules:

$$\frac{\sigma;\ (\Gamma \mid ?_i : A) :: gs \qquad \ldots}{[?_i \mapsto t]\sigma;\ gs' +\!\!+ gs}$$

They should be read as follows: Given a branch with substitution σ, first goal $\Gamma \mid ?_i : A$, and remaining goals gs, refine the first goal to the sub-goals gs' and extend the substitution such that $?_i$ is instantiated with t.

The rules only show the refinement of one branch. To get a full search algorithm, we iteratively refine each search branch with every applicable rule until one without any sub-goals is found. The solution is given by that branch's meta-variable substitution applied (recursively) to the original meta-variable. If no refinement rule is applicable to a branch but it has remaining goals, it is discarded.

Fully applying a term to meta-variables:

$$\frac{(c' : C, gs) = \texttt{AppFull}_\Gamma(c\ ?_i : B)}{\texttt{AppFull}_\Gamma(c : A \to B) = (c' : C, \{\Gamma \mid ?_i : B\} \cup gs)} \qquad \frac{A \neq X \to Y}{\texttt{AppFull}_\Gamma(c : A) = (c : C, \emptyset)}$$

Components:

$$\frac{x : A \in \Gamma}{\texttt{AppFull}(x : A) \in \mathcal{C}_\Gamma}\textsc{CVar}$$

$$\frac{(c : A \times B, gs) \in \mathcal{C}_\Gamma \qquad (c' : C, gs') = \texttt{AppFull}_\Gamma(\texttt{fst}\ c : A)}{(c' : C, gs \cup gs') \in \mathcal{C}_\Gamma}\textsc{Cfst}$$

$$\frac{(c : A \times B, gs) \in \mathcal{C}_\Gamma \qquad (c' : C, gs') = \texttt{AppFull}_\Gamma(\texttt{snd}\ c : A)}{(c' : C, gs \cup gs') \in \mathcal{C}_\Gamma}\textsc{Csnd}$$

Refinements:

$$\frac{\sigma;\ (\Gamma \mid ?_i : A \to B) :: gs}{[?_i \mapsto \lambda x.?_j]\sigma;\ (\Gamma, x : A \mid ?_j : B) :: gs}\textsc{Rlam}$$

$$\frac{\sigma;\ (\Gamma \mid ?_i : A) :: gs \qquad (c : A, gs') \in \mathcal{C}_\Gamma}{[?_i \mapsto c]\sigma;\ gs' +\!\!+ gs}\textsc{Rcomp}$$

$$\frac{\sigma;\ (\Gamma \mid ?_i : A \uplus B) :: gs}{[?_i \mapsto \texttt{inj}_1\ ?_j]\sigma;\ (\Gamma \mid ?_j : A) :: gs}\textsc{Rinj}_1 \qquad \frac{\sigma;\ (\Gamma \mid ?_i : A \uplus B) :: gs}{[?_i \mapsto \texttt{inj}_2\ ?_j]\sigma;\ (\Gamma \mid ?_j : B) :: gs}\textsc{Rinj}_2$$

$$\frac{\sigma;\ (\Gamma \mid ?_i : A \times B) :: gs}{[?_i \mapsto (?_j, ?_k)]\sigma;\ (\Gamma \mid ?_k : B) :: (\Gamma \mid ?_j : A) :: gs}\textsc{Rpair}$$

$$\frac{\sigma;\ (\Gamma \mid ?_i : C) :: gs \qquad (c : A \uplus B, gs') \in \mathcal{C}_\Gamma}{[?_i \mapsto \uplus\texttt{-elim}\ c\ ?_j\ ?_k]\sigma;\ (\Gamma \mid ?_k : B \to C) :: (\Gamma \mid ?_j : A \to C) :: (gs' +\!\!+ gs)}\textsc{R}\uplus\textsc{e}$$

$$\frac{\sigma;\ (\Gamma \mid ?_i : C) :: gs \qquad (c : \bot, gs') \in \mathcal{C}_\Gamma}{[?_i \mapsto \bot\texttt{-elim}\ c]\sigma;\ (gs' +\!\!+ gs)}\textsc{R}\bot\textsc{e}$$

Figure 2.3: Synthesis Algorithm Rules

2.5 Properties of the Algorithm

The algorithm is a *semi-decision* procedure. That is, if the goal type is inhabited, the algorithm will find a solution. However, if it is not, the algorithm may not terminate. For example, consider the non-provable proposition $(A \rightarrow A) \rightarrow A$. After one refinement step, one of the branches will have the goal:

$$\lambda f.?_1 \quad \text{with} \quad f : A \rightarrow A \,|\, ?_1 : A$$

After another step, we get:

$$\lambda f.f\ ?_2 \quad \text{with} \quad f : A \rightarrow A \,|\, ?_2 : A$$

This chain can continue indefinitely. We would expect it to be possible to create a proper decision procedure since validity of propositional logic is well-known to be decidable (or equivalently, the type-inhabitation problem of simply typed λ-calculus is decidable). Indeed, several such algorithms exist (see e.g. the strongly focused sequent calculus LJQ [4]). The algorithm presented here *could* be turned into a proper decision procedure by adding loop detection, but the purpose of the algorithm is to lay the groundwork for an algorithm in dependently typed λ-calculus where type inhabitation is undecidable.

3

Dependently Typed λ-Calculus ($\lambda^?_\Pi$)

The simply typed λ-calculus presented in chapter 2 corresponds to intuitionistic propositional logic and is severely limited in what kinds of propositions it can express. This chapter is concerned with a more expressive *dependently typed* calculus that corresponds to intuitionistic predicate logic[1]. It is based on λ_Π as described in a tutorial implementation in Haskell a by Löh et al. [14]. I have extended their language with meta-variables and added proof-search.

3.1 The Language $\lambda^?_\Pi$

In $\lambda^?_\to$, types and terms live in different worlds. Consequently, types can only specify very rudimentary properties of terms. In dependently typed languages, these worlds are fused so that types may *depend* on terms. Figure 3.1 shows the syntax of $\lambda^?_\Pi$.

$e, \rho ::=$	x	variable
	$\mid \lambda x \to e$	λ-abstraction
	$\mid e\ e'$	(dependent) function application
	$\mid *$	the type of types
	$\mid e : \rho$	term with type annotation
	$\mid \forall x : \rho.\rho'$	dependent function space
	$\mid \_$	hole
	$\mid ?_n$	meta-variable

Figure 3.1: Syntax for $\lambda^?_\Pi$.

The most important difference compared to $\lambda^?_\to$ is the generalization of the function

[1] The calculus is actually unsound because $* : *$ gives rise to Girard's Paradox.

space $A \to B$ to a *dependent* function space, $\forall x : \rho.\rho'$, called a Π-type. The variable x may occur in ρ', making the co-domain *type* dependent on the *value* it is applied to. Importantly, the Π-type should not be confused with a parametrically polymorphic type as found in, e.g., Haskell or System F. In Agda, the Π-type is written $(x : \rho) \to \rho'$.

As an example of how Π-types can express interesting properties, take the proposition "every A that satisfies the predicate P also satisfies the predicate Q." It can be expressed in $\lambda^?_\Pi$ as follows:

$$\forall a : A.\ \forall p : P\ a.\ Q\ a$$

Notice that the dependent function space expresses both universal quantification ($\forall a : A$) and implication ($\forall p : P\ a.\ Q\ a$). In the latter case, the co-domain, $Q\ a$, does not mention p, so it behaves like a non-dependent function space. I will use the arrow notation in this case, writing the previous example as: $\forall a : A.\ P\ a \to Q\ a$.

Other than Π-types, $\lambda^?_\Pi$ differs from $\lambda^?_\to$ by adding type annotations, $e : \rho$. They are practical for dependently typed languages since many terms do not have unique types. For simplicity, I have left out products, co-products, and $\bot$ from the calculus. These can be added with rules analogous to those used for $\lambda^?_\to$.

Further, holes (denoted by an underscore, _) are kept separate from meta-variables (denoted $?_n$). The reason is that meta-variables may occur more than once in a term or type, and are therefore abstracted over the context. For example, reducing $(\lambda x \to f\ (\lambda y \to x)\ x)\ ?_0$ yields $f\ (\lambda y \to ?_0)\ ?_0$. Notice that these occurrences are in different contexts. Hence, holes are for "user input" and the type checker replaces them with fresh meta-variables applied to all the variables in the context. For example, it will replace $\lambda x \to \ \_ : A \to B$ with $\lambda x \to ?_0\ x : A \to B$ for a new meta-variable $?_0 : A \to B$.

Figure 3.2 gives the typing rules for $\lambda^?_\Pi$. Following the presentation by Löh et al., the rules are marked with arrows pointing up ($:_\uparrow$) if the rule is used for type inference, and down ($:_\downarrow$) if it is used for checking. The type in the rules should be thought of as input for the checking rules and output for the inference rules. I have marked the parts that differ from λ_Π in gray. Since holes are substituted for new meta-variables, the rules technically produce new terms, but I have left this implicit in all the rules but Hole for readability. The reduction relation, $\Downarrow$, is a big-step reduction and $v[x \mapsto e]$ denotes substitution of e for all free occurrences of x in v.

Reduction:

$$\frac{e \Downarrow v}{e : \rho \Downarrow v} \qquad \frac{}{* \Downarrow *} \qquad \frac{\rho \Downarrow \tau \qquad \rho' \Downarrow \tau'}{\forall x : \rho.\rho' \Downarrow \forall x : \tau.\tau'} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e \Downarrow \lambda x \to v \qquad v[x \mapsto e'] \Downarrow v'}{e\ e' \Downarrow v\ v'} \qquad \frac{}{?_n \Downarrow ?_n}$$

Typing rules:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}\textsc{Var} \qquad \frac{}{\Gamma \vdash * :_\uparrow *}\textsc{Star} \qquad \frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\textsc{Ann}$$

$$\frac{\Gamma \vdash e :_\uparrow \forall x : \tau.\tau' \qquad \Gamma \vdash e' :_\downarrow \tau \qquad \tau'[x \mapsto e'] \Downarrow \tau''}{\Gamma \vdash e\ e' :_\uparrow \tau''}\textsc{App}$$

$$\frac{\Gamma \vdash e :_\uparrow \tau' \qquad \tau \simeq \tau'}{\Gamma \vdash e :_\downarrow \tau}\textsc{Chk} \qquad \frac{\Gamma, x : \tau \vdash e :_\downarrow \tau'}{\Gamma \vdash \lambda x \to e :_\downarrow \forall x : \tau.\tau'}\textsc{Lam}$$

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma, x : \tau \vdash \rho' :_\downarrow *}{\Gamma \vdash \forall x : \rho.\rho' :_\uparrow *}\textsc{Pi} \qquad \frac{\tau = \texttt{LookupMeta}(?_n)\ \Gamma}{\Gamma \vdash ?_n :_\uparrow \tau}\textsc{Meta}$$

$$\frac{?_n = \texttt{NewMeta}(\forall \bar{x} : \Gamma.\tau)}{\Gamma \vdash \_ :_\downarrow \tau \longmapsto ?_n\ \Gamma}\textsc{Hole}$$

Figure 3.2: Typing- and reduction rules for $\lambda^?_\Pi$. In the rule HOLE, the hole (_) is substituted for a new term consisting of a new meta-variable, $?_n$, applied to all the arguments in Γ.

An important difference from $\lambda^?_\to$ is that semantic equality ($\simeq$) of types no longer coincides with syntactic equality. This is just like in everyday mathematics where $1 + 1$ is considered to be equal to 2 even though they are syntactically distinct. Consequently, reduction of terms becomes an integral part of type checking and synthesis, whereas they were mostly inconsequential in $\lambda^?_\to$. In the original λ_Π, equality is checked by fully reducing both terms and then checking syntactic equality[2]. However, this method is insufficient when the language includes meta-variables. Therefore, $\lambda^?_\Pi$ uses conversion checking $\simeq$. For example, the rule CHK is changed as follows:

[2]This only works because they use a locally nameless representation, so no α-conversion is necessary.

$$\lambda_\Pi\colon \frac{\Gamma \vdash e :_\uparrow \tau}{\Gamma \vdash e :_\downarrow \tau} \qquad \lambda^?_\Pi\colon \frac{\Gamma \vdash e :_\uparrow \tau' \qquad \tau \simeq \tau'}{\Gamma \vdash e :_\downarrow \tau}$$

Conversion checking works through unification. There are many unification algorithms (e.g., [16]) and the language should work with (essentially) any one of them. Therefore, I do not specify conversion checking but give an informal account instead. The idea is that unification takes a set of equality constraints and tries to find substitutions for any meta-variables that make all constraints true. However, unification does *not* try to generate new terms, it only uses what can be derived from the equality constraints. For a set of equality constraints, a unification algorithm can give three different results: (1) there is no solution, (2) a complete solution is found, (3) there are remaining constraints. Remaining constraints after type checking and synthesis are considered to be errors.

Both type checking and synthesis need to keep track of meta-variables and equality constraints. For this, I introduce signatures, Σ:

$\Sigma ::= \varepsilon$	Empty context
$\mid [?_n : \tau]\Sigma$	Add meta variable
$\mid [?_n \mapsto t]\Sigma$	Add meta-variable instantiation
$\mid [s = t]\Sigma$	Add equality constraint

The signature is modified by `NewMeta` and conversion checking ($\simeq$) during type-checking. I will write, e.g. $?_n = \texttt{NewMeta}_\Sigma(\tau) \Longrightarrow \Sigma'$ to denote those modifications. However, I have left the signatures implicit in the typing rules to make them more readable. Also, I have left out the rules for validity of contexts. They can be found in the original exposition of λ_Π [14]. For more details about meta-variables, see Norell's thesis about Agda [17]. Because $\lambda^?_\Pi$ does not contain products, sums, natural numbers, identity types etc., it is hard to give accessible example programs. Readers who are new to dependently typed programming may therefore want to skip ahead to chapter 4 for some examples in Agda.

3.2 Synthesis Algorithm

The underlying principle for synthesis is the same as for $\lambda_\rightarrow$: Successively refine the goal (represented by a meta-variable) by speculatively applying introduction rules and using components derived from the context using elimination rules. However, there are some important differences:

3.2.1 Unification

Checking that two types are equal, which is a key step in trying to use a component to fill a goal, involves unification. As mentioned previously, unification also involves

reduction. This makes synthesis more expensive, and we should expect it to be the costliest part of the algorithm.

Also, recall that unification may leave unsolved constraints. One approach is to terminate a search branch if there are unsolved constraints. Another approach is to wait until the synthesis is complete to check for unsolved constraints. The latter gives a larger space of possible solutions but can be slower. If the former approach is used, it is conceivable that the unsolved constraints could guide the search, e.g. by changing the order in which sub-goals are solved, but I have not explored these possibilities.

3.2.2 Dependent sub-goals

For simple types, all sub-goals are independent of each other. This is not the case in $\lambda^{?}_{\Pi}$. Assume that we have the dependent identity function $\mathrm{id} : \forall A : *.\ A \to A$ and consider the following term (for readability, I do not write the meta-variables abstracted over the context):

$$\begin{aligned} \mathrm{id}\ ?_0\ ?_1 \qquad & \text{where} \\ \varepsilon &\vdash ?_0 : * \\ ?_0 : * &\vdash ?_1 : ?_0 \end{aligned}$$

Here, the type of $?_1$ depends on $?_0$. On the one hand, such dependencies make it possible for a solution to one sub-goal to cause a different sub-goal to be unsolvable. For example, an empty type is a solution for $?_0$ but makes it impossible to solve $?_1$. This means that an algorithm may need to backtrack. Luckily, the iterative search we use already has back-tracking "built in." However, it still makes the search procedure slower and sub-goals cannot easily be solved in parallel. On the other hand, solving one sub-goal can determine a solution for a different one. For example, if $?_1$ is solved by $b : B$, we get the solution $B : *$ for $?_0$ for free through unification.

3.2.3 Arity

In $\lambda_{\to}$, a function's arity is easily determined syntactically from its type. However, in $\lambda^{?}_{\Pi}$, a function may take a different number of arguments depending on the *value* of its previous arguments. For example, many arguments does g take?

$$g : \forall f : * \to * \to *. \forall A : *. f\ A\ (A \to A)$$

The answer depends on the value of f:

$$\begin{aligned} g\ (\lambda A\ x\ y \to x) &: \forall A : *. A \\ g\ (\lambda A\ x\ y \to y) &: \forall A : *. A \to A \end{aligned}$$

This has consequences for synthesis. Recall that the algorithm for $\lambda_{\to}$ applies each component to meta-variables as long as its type is a function space. With this

strategy, not all valid components will be generated. I believe this case is rare enough to keep using the same strategy despite its limitations.

3.2.4 The Algorithm

This section describes a full synthesis algorithm for $\lambda^?_\Pi$. It is –by design– very similar to the algorithm for $\lambda^?_\to$ given in section 2.4. I use the following notation:

$\Gamma \vert ? : \tau$	Goal: Find a term of type τ in context Γ
ε	Empty list (of goals)
$g :: gs$	Cons: List with head g and tail gs
$gs +\!\!+ gs'$	Concatinate lists
$\Sigma; gs$	A search branch is a signature and a set of goals

As before, the algorithm has two parts. The first part generates components from the context. Since the only type constructor in $\lambda^?_\Pi$ is the function space (there are, e.g., no products), the only rules are to fully apply the variables in the context to new meta-variables. The application is done by $\texttt{AppFull}_{\Sigma;\Gamma}(c : \tau) = (c' : \tau', gs) \Longrightarrow \Sigma'$ which takes a term, $c : \tau$, and produces a new term, $c' : \tau'$, together with a set of new meta-variables, gs, and an updated signature Σ'. Notice that `AppFull` reduces the type before checking if it is a Π-type. The component rules just apply `AppFull` to every variable in the context with the appropriate book-keeping for the signatures.

The second part is the refinement rules. They have the following form:

$$\frac{\Sigma;\ (\Gamma \vert ?_i : \tau) :: gs \qquad \ldots}{\Sigma';\ gs'}$$

They should be read as follows: A branch with signature Σ, first goal $\Gamma \vert ?_i : \tau$, and remaining goals gs can be refined to a new branch with signature Σ' and goals gs'. The rule RLam refines by λ-abstraction and the rule RComp refines by using a component.

Like for $\lambda^?_\to$, the rules only show the refinement of one goal of a search branch. To get the full algorithm the refinement rules are applied to each search branch in turn, producing new search branches. This continues iteratively until a branch has no remaining goals. The solution term is then the instantiation of the original meta-variable as given by the branch's signature.

Fully applying a term to meta-variables:

$$\frac{\begin{array}{l}\tau \Downarrow \forall \tau_1.\tau_2 \\ ?_n = \texttt{NewMeta}_{\Sigma}(\forall \bar{x} : \Gamma.\forall x{:}\tau_1.\tau_2) \quad \Longrightarrow \Sigma' \\ (c'{:}\tau', gs) = \texttt{AppFull}_{\Sigma';\Gamma}(c\ ?_n : \tau_2[x \mapsto ?_n]) \quad \Longrightarrow \Sigma''\end{array}}{\texttt{AppFull}_{\Sigma;\Gamma}(c{:}\tau) = (c'{:}\tau', (\Gamma|?_n) :: gs) \Longrightarrow \Sigma''}$$

$$\frac{\tau \Downarrow \tau' \qquad \tau' \neq \forall x{:}\tau_1.\tau_2}{\texttt{AppFull}_{\Sigma;\Gamma}(c{:}\tau) = (c{:}\tau, \varepsilon) \Longrightarrow \Sigma}$$

Components:

$$\frac{\Gamma = \varepsilon}{\texttt{Comps}(\Sigma;\Gamma) = \emptyset \Longrightarrow \Sigma}\textsc{CEmpty} \qquad \frac{\begin{array}{l}\Gamma = \Gamma', x{:}\tau \\ (c{:}\tau', gs) = \texttt{AppFull}_{\Sigma;\Gamma}(x{:}\tau) \quad \Longrightarrow \Sigma' \\ \mathcal{C} = \texttt{Comps}(\Sigma';\Gamma') \quad \Longrightarrow \Sigma''\end{array}}{\texttt{Comps}(\Sigma;\Gamma) = \{(c{:}\tau', gs)\} \cup \mathcal{C} \Longrightarrow \Sigma''}\textsc{CVar}$$

Refinements:

$$\frac{\Sigma;\ (\Gamma|?_i : \forall x{:}\tau.\tau') :: gs \qquad ?_j = \texttt{NewMeta}_{\Sigma}(\forall \bar{x}{:}\Gamma.\tau) \Longrightarrow \Sigma'}{[?_i \mapsto \lambda x \to ?_j]\Sigma';\ (\Gamma, x{:}\tau|?_j{:}\tau') :: gs}\textsc{RLam}$$

$$\frac{\Sigma;\ (\Gamma|?_i{:}\tau) :: gs \qquad (c{:}\tau', gs') \in \texttt{Comps}(\Sigma;\Gamma) \Longrightarrow \Sigma' \qquad \tau \simeq_{\Sigma'} \tau' \Longrightarrow \Sigma''}{[?_i \mapsto c]\Sigma'';\ gs' +\!\!+ gs}\textsc{RComp}$$

Figure 3.3: Synthesis rules for $\lambda^{?}_{\Pi}$.

3.2.5 Properties of the Algorithm

Recall that the algorithm for $\lambda^{?}_{\to}$ was a semi-decision procedure and that it is possible to modify it to be a full decision procedure. This is not the case for the algorithm described here. Unlike for simply typed λ-calculus, the type inhabitation problem is not decidable for dependently typed λ-calculus [15]. This should be expected since it corresponds to predicate logic where satisfiability is undecidable. That is, the algorithm for $\lambda^{?}_{\Pi}$ only works sometimes. Also, re-iterating the point from section 3.2.3, variables whose arities depend on their parameters may cause the algorithm to fail.

4

Agda and Mimer

Agda, like $\lambda^?_\Pi$, is dependently typed. However, unlike $\lambda^?_\Pi$ which is a simple calculus, Agda is a full-featured programming language and includes things like user-defined data types, modules, universe polymorphism, and a much larger range of syntactic constructs. Nonetheless, the theoretical underlying calculus is similar. Agda is often used as a proof-assistant through the Curry-Howard correspondence (as described in chapters 2 and 3) by interpreting types as propositions, programs as proofs and type checking as checking the correctness of a proof.

This chapter describes the main contribution of this thesis[1], Mimer[2]. It is a tool for synthesizing small programs (proofs) in Agda. As mentioned in the introduction, Agda already has a synthesis tool called Agsy that has not kept up with the evolution of Agda. Mimer is thus a possible replacement for Agsy.

4.1 Aim and Design

Mimer is typically invoked as an interactive command in a text editor. The most common use-case is to quickly fill in small details of proofs. Mimer's default time-out is set to one second, allowing programmers to routinely invoke the tool to see if it can solve a goal before they start to do it manually, thus saving time. Hence, Mimer is designed to handle many small synthesis problems rather than complex ones. Also, Mimer does not perform case splits and does not generate new auxiliary functions.

Further, Mimer is designed to be robust to changes and easy to maintain. Agda is a constantly evolving research language and some features such as Cubical Agda and subtyping can be selectively enabled. Consequently, a synthesis system that is highly specialized to the current language features is likely to go out of sync with the language as it changes, as happened with Agsy. In the hope of making Mimer more stable in this regard, it reuses as much of the compiler as possible. In particular, it reuses data types for representing types and terms, functions for term conversion and reduction (including unification), and management of meta-variables. Also, Mimer

[1] Mimer is available online at `https://github.com/Lukas-Skystedt/agda`

[2] In Norse mythology, Mimer (or Mímir) guards a well of wisdom. After being decapitated, the god Odin keeps Mimer's head and it recites knowledge to him.

is a part of the same code base as the compiler, so changes to, e.g., data types used by Mimer will cause compilation errors for the compiler developers, forcing them to make necessary changes to Mimer in parallel with the compiler itself. Additionally, Mimer mostly treats all types uniformly and has no special logic for, e.g., natural numbers or setoids.

Unlike $\lambda^?_\to$ and $\lambda^?_\Pi$, there is no full specification of Agda so I do not give a formal description of the algorithm. Instead, sections 4.2.1–4.2.6 give a high-level description of how the algorithm handles various language features that are different from the previously mentioned calculi.

4.2 Mimer

When Mimer is invoked, the program has already passed through the type checker, so it works directly on well-typed terms. When invoked, Mimer goes through three stages:

Setup. First, it collects information that is used during the search: Definitions of data types, records, external functions, and postulates as well as let-bound variables.

Search. The actual search algorithm, like for $\lambda^?_\to$ and $\lambda^?_\Pi$, has two parts. The first part generates components from the definitions collected at setup and local variables. The second part refines goals top-down using the components, λ-abstraction and constructors (section 4.2.1).

Post-processing. Like the previously described algorithms, Mimer generates terms on η-long form. However, to make the resulting code more readable, it η-reduces the solution (handled by pre-existing functions in the compiler), producing f instead of $\lambda\ x\ \to\ f\ x$. If necessary, it also brings hidden local variables into scope; see section 4.2.3.

4.2.1 Data Types and Records

Agda allows the programmer to define parameterized and indexed inductive data types. For example, a binary sum type (co-product) similar to those built into $\lambda^?_\to$ can be defined as follows:

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

The syntax should look familiar to readers who are familiar with Haskell's GADT syntax. It defines a new type, `_⊎_`, parameterized by two types. It has two constructors, inj_1 and inj_2. Unlike in $\lambda^?_\to$, there is no explicit eliminator (`⊎-elim`). Instead, Agda eliminates data types through pattern matching on the left-hand side of function definitions. The aforementioned eliminator can be defined in terms of pattern matching:

```
⊎-elim : (A B C : Set) → A ⊎ B → (A → C) → (B → C) → C
⊎-elim A B C (inj₁ a) f g = f a
⊎-elim A B C (inj₂ b) f g = g b
```

Mimer searches for terms of these data types similarly to how the algorithm for $\lambda^{?}_{\rightarrow}$ searches for sums and products: It speculatively refines the goal with each constructor fully applied to new meta-variables.

Inductive data types can also be used to define products ($\times$). However, Agda also has records for defining data types with only one constructor:

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
```

This introduces a new type, `_×_` , with a constructor, `_,_`. With records, we *do* get eliminators (projections); in this case, `fst` and `snd`. As an example, we can prove that conjunction is commutative (recall that products correspond to logical conjunction):

```
×-commutative : (A B : Set) → A × B → B × A
×-commutative A B AxB = snd AxB , fst AxB
```

Mimer treats record constructors the same way as data constructors. However, it handles record projections similarly to how the algorithm for $\lambda^{?}_{\rightarrow}$ handles product projections: Whenever a component has a record type, it applies its projections to produce new components. It continues this recursively for the new components. There is, however, an exception. Records may be recursive, either inductive or coinductive, meaning a field can have a type that involves the record type itself. For instance, consider the coinductive infinite stream type[3]:

```
record Stream (A : Set) : Set where
  coinductive
  field
    elem : A
    next : Stream A
```

The projection `next` can be applied an unlimited number of times. Hence, Mimer detects loops when generating components. If a component has a record type that has already been seen in the derivation of the component, it does not apply any more projections. Importantly, only the record name itself is considered, not the record with its parameters since this could still lead to an infinite number of projections. To summarize, data constructors and record constructors are used for refining goals while record projections are used when generating components.

The method of only applying projections to components that are known to be of record type is considerably faster than refining goals with projections speculatively. However, it is sometimes too restrictive. Similarly to how a function's arity cannot

[3]coinductive records requires the option `-guardedness`

always be determined (see section 3.2.3), it is not always possible to know that a projection is applicable to a component because its type is blocked on a meta-variable. Therefore, Mimer has an option (disabled by default) to try projections speculatively (the same way as external function definitions) as well, but at a higher cost (see section 4.2.8).

Finally, if the goal type is Set, Mimer uses the type constructors (e.g. `_×_`) themselves as components. However, in practice, it is rare that Mimer has to explicitly synthesize terms of type Set because those parameters can often be inferred by unification.

4.2.2 User Language and Internal Language

Agda consists of two languages: a user language that is designed to be convenient for human programmers, and an internal language that is considerably smaller and well suited for manipulation by computers. $\lambda^{?}_{\Pi}$ is (essentially) a fragment of Agda's internal language. It is the type checker's job to translate the user language to the internal language in which Mimer performs its synthesis. Two language features that are unique to the user language are implicit arguments (section 4.2.3) and let-bound variables (section 4.2.4).

4.2.3 Implicit Arguments

In practice, Agda functions often have many parameters, often due to type-variables being explicit parameters when using dependent types. For example, the polymorphic identity function can be written as follows:

```
id : (A : Set) → A → A
id A x = x
```

In Agda, the type of types is Set, corresponding to $*$ in $\lambda^{?}_{\Pi}$. Writing all the parameters explicitly can be cumbersome. Therefore, the user language has *implicit* arguments that the type checker infers automatically. In fact, implicit arguments is one of the reasons Agda has meta-variables as the inference mechanism is based on them. Implicit arguments are written with braces ({ }) around them. Making A implicit, the identity function can be written as follows:

```
id₂ : {A : Set} → A → A
id₂ x = x
```

However, a problem arises when Mimer synthesizes a term that uses an implicit argument in a position where it must be explicit. For example, consider the following (somewhat contrived) example:

```
data Box (A : Set) : Set where
  box : {a : A} → Box A

unbox : {A : Set} → Box A → A
unbox box = ?
```

Mimer might synthesize the term n — the local variable of type Nat given as an implicit argument to the constructor box. However, in the user language, n is not in scope. Therefore, Mimer post-processes the solution term by finding all variables that are out of scope in the user language and adds them to the pattern on the left-hand side of the function clause. A detail here is that finding what variables occur in the solution term cannot be determined simply by tracking what components were used since meta-variables can be instantiated with a variable during unification. The result of invoking Mimer on the previous example would thus be the following:

```
unbox : {A : Set} → Box A → A
unbox box = ?
```

4.2.4 Let-Bound Variables

The user language has let-bindings. They are very likely to be useful during synthesis — why else would the user have defined them? However, let-bindings are inlined during type checking and are not present in the internal language. Hence, Mimer uses the terms from the right-hand side of the bindings as components during the search. The unfortunate consequence of inlining is that every occurrence of a let-bound variable will appear as the full right-hand side of the binding rather than as the name of the bound variable in a synthesized term. Consider, for example, the following synthesis problem:

```
use-let : SomeType
use-let = let x = long-and-complex-expression
              y = longer-and-more-complex-expression
          in ?
```

Mimer might find a solution that involves x and y, but the inlined terms would be displayed instead:

```
use-let : SomeType
use-let = let x = long-and-complex-expression
              y = longer-and-more-complex-expression
          in final-step long-and-complex-expression longer-and-more-complex-expression
```

Therefore, as a part of post-processing, Mimer performs back-substitution of let-bound variables. Hence, it would produce the more user-friendly version:

```
use-let : SomeType
use-let = let x = long-and-complex-expression
              y = longer-and-more-complex-expression
          in final-step x y
```

4.2.5 Universe Polymorphism

Most of the Agda types presented so far have had the type, also called sort or universe, Set. In $\lambda^{?}_{\Pi}$, the type of types is $*$ and it has itself as its type, $* : *$. This is simple, but unsound since it can be used to express Girard's paradox — a

type-theory version of Russel's paradox for set theory. To avoid paradoxes, Agda has an infinite universe hierarchy where Set : Set_1 and Set_1 : Set_2 etc. To allow programmers to write functions that work in any universe (universe-polymorphic functions), it has a special sort called Level with a zero-element, successor, and join operator[4]. To showcase universe polymorphism, I take the opportunity to define the dependent product type, Σ:

```
record Σ {a b : Level} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    fst : A
    snd : B fst
```

The dependent product is a pair where the type of the second component can depend on the value of the first component. Recall that Π-types correspond to universal quantification (∀). Similarly, Σ-types correspond to existential quantification (∃). For example Σ Nat (λ m → n ≡ 2 * m) states that n is even[5]. The definition above is universe-polymorphic. The component types may belong to universes of different levels (a and b) and the pair type itself belongs to the highest of these two universes ($a \sqcup b$).

Mimer keeps components used for levels and universes separate from the components used for other types. Other than that, the method for synthesizing them is essentially the same as for regular types. However, in the vast majority of cases, levels and universes are solved by unification, so it is rare that Mimer has to explicitly synthesize them.

4.2.6 Recursion and Termination

In Agda, functions may call themselves recursively. In terms of proofs, a recursive calls corresponds to using the induction hypothesis in a proof by structural induction. However, Agda does not allow all recursive calls but imposes a termination condition. If it did not, every type would be inhabited by an infinitely looping term:

```
absurd : ⊥
absurd = absurd
```

The termination condition is that, throughout the call graph, some argument must consistently become structurally smaller. Agda considers an argument to be structurally smaller if it originates from inside a constructor in a pattern match on the left-hand side of a definition. For example, the following definition of addition terminates because in the recursive call to add, the first argument, n, is structurally smaller than the input suc n.

```
add : Nat → Nat → Nat
add zero m = m
add (suc n) m = suc (add n m)
```

[4]The join operator gives the max (least upper bound) of two levels.

[5]A natural number n is even if there exists a natural number m such that $n = 2 * m$

The complete termination rules are rather complex since they consider the entire call-graph across mutually recursive functions and takes co-induction into account. To make matters worse, it is hard to reuse the compiler's termination checker because it does not leave annotations. For this reason, Mimer uses a simple approximation for termination when synthesizing recursive calls. In the setup phase, it finds all local variables that could be candidates for a "shrinking" argument in a recursive call. It then generates the components consisting of recursive calls with at least one of the candidate variables as an argument. Making calls to mutually recursive functions is left as future work.

This approximation is too lax so Mimer can generate non-terminating terms. However, Agda's soundness is unaffected since the solution is always checked afterwards. In practice, the approximation seems to work well — I have only encountered the defect in contrived examples. Nonetheless, I think it would be beneficial to update the compiler's termination checker so that it can be reused for synthesis, especially for code using the Sized Types extension.

4.2.7 Caching Components

Recall that Mimer generates components from the context and external definitions by applying new meta-variables and record projections to them. It is wasteful to compute all the components for every branch refinement. Therefore, Mimer caches the generated components so that they can be reused for future refinements. There are two restrictions to when a cached component may be used.

Firstly, a component is only valid in the context it was generated in. This is because, like in $\lambda^{?}_{\Pi}$, meta-variables are abstracted over the context. The context can change either because Mimer refines a goal with a λ-abstraction, or because Agda has pruned a meta-variables context to make all its occurrences valid. If Mimer enters a new context, it regenerates all components.

Secondly, if a refinement with a component is successful, the components that mention the same meta-variables as that component must be regenerated to prevent unwanted interference between sub-goals.

4.2.8 Guiding Search using Branch Costs

Mimer uses a cost metric as a mechanism to heuristically guide the search. For example, it is more likely that a solution to a given goal will contain local variables than some particular external definition. To reflect this observation, Mimer associates each search branch with a cost. When applying a refinement, it increases the cost of that branch depending on what refinement it was. For instance, it will assign a higher cost to uses of external functions than local variables. Mimer keeps the branches in a priority queue ordered by the costs. At each step of the search, instead of refining every open branch, it selects the one with the lowest cost from the queue.

The costs are highly configurable and may depend on what kind of component it is (used projections, constructors, local variables, let-bound variables, etc.), how many open meta-variables it contains, and more. It can also assign a higher cost to reusing the same symbols many times. This stems from the intuition that speculatively refining with the same thing over and over again is unlikely to lead anywhere. For example, a goal of type Nat can be refined to suc ? which in turn can be refined to suc (suc ?) and so on.

The costs only apply to explicit refinements, not sub-goals that are solved by unification. Finding good values for the cost parameters remains as future work.

4.2.9 Mimer's User Interface

Mimer's user interface is similar to Agsy's. Typically, Agda programmers would invoke Mimer through their text editor by placing their cursor inside a hole and typing a keyboard shortcut. Additionally, Mimer accepts extra options through an argument string typed directly into the hole, similar to terminal command options. That way, the user can specify, e.g., a time limit (default is one second), whether to use projections speculatively, and what components to use. It is expensive to try a large number of components, so using everything in scope is generally inadvisable. By default, Mimer will only use data types, data constructors, record types, record constructors, local variables, let-bindings, and things bound in where-clauses. There are options to include everything declared in the current module or everything in scope.

Further, Mimer has more options available internally, that, at the time of writing, cannot be configured outside of the source-code. These include the cost parameters, restrictions to what projections to generate from the available components, and whether or not to speculatively use record projections.

4.2.10 Advanced Agda Features

Agda has a range of advanced language features and language extensions. For example, Cubical Type Theory, Flat Modality, and Sized Types. Mimer has not been tested with these extensions.

4.3 Results

To compare Mimer to Agsy, it has been tested on two sets of synthesis problems. The first problem set is Agsy's own test suite, consisting of 33 problems. The second set is developed specifically to test Mimer, and consists of 44 problems[6]. Tables 4.1 and 4.2 show the running time for the two synthesis tools on the two problems sets.

[6]Both problem sets are available in Mimer's GitHub repository, `https://github.com/Lukas-Skystedt/agda`

Table 4.1: Running times for Agsy and Mimer for the problems in Agsy's test suite. A dash (–) signifies that no solution was found.

#	Agsy (ms)	Mimer (ms)
1	1	36
2	2	2
3	2	0
4	3	14
5	3	23
6	3	19
7	4	–
8	4	–
9	4	21
10	4	7
11	5	–
12	5	21
13	6	12
14	6	–
15	7	62
16	9	51
17	10	16
18	11	76
19	11	–
20	11	261
21	14	–
22	15	–
23	20	84
24	20	66
25	27	–
26	29	–
27	32	–
28	34	–
29	67	–
30	73	1937
31	80	–
32	291	157
33	1102	–

Table 4.2: Running times for Agsy and Mimer for the problems in Mimer's test suite. A dash (–) signifies that no solution was found.

#	Agsy (ms)	Mimer (ms)
34	1	0
35	6	0
36	1	0
37	5	0
38	–	0
39	–	0
40	1	1
41	1	1
42	3	1
43	–	1
44	1	2
45	1	2
46	–	2
47	–	3
48	–	3
49	1	4
50	–	4
51	3	5
52	1	5
53	–	5
54	7	6
55	2	6
56	–	7
57	2	7
58	1	8
59	4	9
60	–	10
61	–	10
62	4	12
63	3	15
64	7	17
65	7	22
66	11	22
67	1	24
68	–	25
69	4	25
70	20	27
71	–	34
72	6	37
73	–	37
74	–	46
75	15	49
76	10	76
77	17	76

Mimer failed to solve 14 of the 33 problems in Agsy's suite. Of these, it failed 10 because they require case splitting, 1 because it timed out, 2 because it does not have equality reasoning, and 1 for unknown reasons. Agsy failed to solve 16 of the 44 problems in Mimer's test suite for the following reasons (Mimer can solve all the associated examples):

1. A function was defined with co-patterns.

```
copattern : ⊤ × ⊤
_×_.fst copattern = ?
_×_.snd copattern = ?
```

2. The goal type had a sort other than Set.

```
constant-level : Level
constant-level = ?
```

3. Agsy timed out (likely due to solving sub-goals in the "wrong" order).

```
big-Σ : Σ Nat (λ n → n ≡ 100000)
big-Σ = ?
```

4. The solution requires the use of let-bound variables. Note that, by default, the synthesizers will not use external definitions, so l cannot be used directly.

```
postulate
  L : Set
  l : L
use-let-binding : L
use-let-binding = let solution = l in ?
```

5. The solution requires lemmas given in where-clauses.

```
postulate WhereTest1 : Set
use-where-postulate : WhereTest1
use-where-postulate = ?
  where postulate where-clause : WhereTest1
```

6. The solution requires the use of postulates. The option `-m` is given in the goal ({!-m!}) to allow the synthesizers to use external definitions in the module.

```
data ⊥-post : Set where
postulate absurd-post : ⊥-post
use-postulate : ⊥-post
use-postulate = {!-m!}
```

7. The goal type was Set.

```
data SomeSet : Set where
set : Set
set = ?
```

8. The solution depends on hidden variables. In this case, Agsy finds a solution that would be correct if x were in scope. Mimer alters the left-hand side of the function so that x is in scope.

```
use-hidden : (A : Set) → {x : A} → A
use-hidden A = ?
```

Overall, Agsy is faster and can solve goals using case-splitting and equality reasoning. However, Mimer supports a wider range of language features.

4.4 Performance Profiling

Mimer's main drawback compared to Agsy is that it is considerably slower. To improve the performance, I have tried to determine where Mimer spends most of its time, but mostly failed. The reason is that Mimer is written in Haskell which has a call-by-need evaluation order. Therefore, measuring the time spent on a computation cannot be performed the same way as in strictly evaluated languages. For example, consider the following attempt at measuring time (written in pseudo-Haskell):

```
startTime ← getTime
result    ← theComputation
stopTime  ← getTime
```

The value of `stopTime − startTime` would likely not give the desired result because `theComputation` would not be fully evaluated. I have tried forcing the computation to fully evaluate, but it changed the performance characteristics enough to make the measurements unreliable. I have also tried to use GHC's[7] profiling tools, but have not found any usable results.

[7]GHC is a compiler for Haskell.

4.5 Future Work

Mimer can be improved in many ways. I believe the most important ones are the following:

Optimizations. Since Mimer is much slower than Agsy, I think optimizing it is the most important next step. The cost parameters can probably be tweaked using standard statistical methods. The harder problem is finding a good way of profiling Mimer to reveal where the largest amount of time is spent.

Case splitting. Agsy has support for case-splitting, which is sometimes useful. However, it should be noted that since Agda only performs case-splits on the left-hand side of function definitions, this feature is less useful than in languages that have, e.g., case-of expressions. Agda already contains machinery for computing case splits, and Mimer can compute candidates for case splitting. Hence, adding case-splits to Mimer should be a matter of connecting the two. The simplest strategy for case-splitting is to naively split on every possible argument (up to some fixed depth). A more sophisticated strategy would be to detect when reduction is stuck on a variable and only then perform a case split.

Equality reasoning. It is very common to write equality proofs in Agda. Special equality reasoning logic can be much more efficient than the general term-search algorithm that Mimer uses. Agsy has some support for equality reasoning, and Mimer could be extended similarly.

Improve termination metrics. As mentioned in section 4.2.6, Mimer uses a simple heuristic for generating recursive calls. It would be an improvement if Agda's termination checker could efficiently be used instead, especially when using Sized Types. Also, Mimer does not currently generate calls to mutually recursive functions to reduce the risk of non-terminating solutions.

4.6 Related Work

Synthesis of both programs and proofs are active research areas and there are a wide range of methods for both formal calculi and real-world languages.

There are other proof assistants than Agda. Coq and Lean are two of examples that both have dependently typed core calculi (versions of the Calculus of constructions). However, unlike in Agda, the user typically does not write program terms directly. Instead, they use programs, called *tactics*, to generate or transform the core language terms. Mimer is a general tactic and fills a similar role to the various auto tactics in Coq. The programming language that is probably closest to Agda is Idris [2]. It has a synthesis tool that fills a similar role to Mimer.

In logic, natural deduction is very close to the calculi considered in this report. An alternative is to use sequent calculi, which can be better suited to search. Dyckhoff

and Lengrand have designed focused sequent calculi for proof search in intuitionistic logic [4] and pure type systems [11]. Further, logic-based search tools, such as SAT solvers, that rely on the *semantics* of logical formulae are common, but work very differently from those that work in proof calculi.

Program synthesis is not only used in languages with a logical interpretation. Application of synthesis include superoptimization of program fragments, automatic bug detection, and automatic data-wrangling in spreadsheets [6]. In these applications, as opposed to proof search, the goal is to synthesize programs with some desired behavior. Therefore, many tools take extra forms of specification as input, e.g., pre- and postconditions, examples of inputs and desired outputs, a grammar defining the search space [1, 20], or a reference implementation [6]. In Synquid [19], the specification is given as a refinement type. Its search algorithm is type-directed, but also relies on an SMT[8] solver for the refinements. Further, there are other approaches to type-directed search than the top-down refinement, such as theory exploration by random testing [3] and abstraction refinement [7]. Gulwani et al. [6] provides an overview of modern program synthesis.

Other than Agsy [13, 12], proof automation for Agda has been implemented as a library in Agda using reflection [9], by embedding SMT queries that are fed to an external solver at compile-time [10], and via integration with an equational solver [5].

[8] An SMT solver is a SAT solver extended with theories for, e.g., numbers and data structures.

Bibliography

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[2] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[3] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, pages 392–406. Springer, 2013.

[4] Roy Dyckhoff and Stéphane Lengrand. Ljq: a strongly focused calculus for intuitionistic logic. In *Conference on Computability in Europe*, pages 173–185. Springer, 2006.

[5] Simon Foster and Georg Struth. Integrating an automated theorem prover into agda. In *NASA Formal Methods Symposium*, pages 116–130. Springer, 2011.

[6] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[7] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[8] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[9] Pepijn Kokke and Wouter Swierstra. Auto in agda. In *International Conference on Mathematics of Program Construction*, pages 276–301. Springer, 2015.

[10] Wen Kokke. Schmitty. `https://github.com/wenkokke/schmitty/`.

[11] Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A focused sequent calculus framework for proof search in pure type systems. *Log. Methods Comput. Sci.*, 7(1), 2011.

[12] Fredrik Lindblad. Higher-order proof construction based on first-order narrowing. *Electronic Notes in Theoretical Computer Science*, 196:69–84, 2008. Proceedings of the Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2007).

[13] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in agda. In *International Workshop on Types for Proofs and Programs*, pages 154–169. Springer, 2004.

[14] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*, 102(2):177–207, 2010.

[15] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[16] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991.

[17] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[18] Benjamin Pierce. *Types and Programming Languages*. MIT Press, Cambridge, 2002.

[19] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[20] Armando Solar-Lezama. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*, pages 4–13. Springer, 2009.