Constraint-Based Type-Directed Program Synthesis

Peter-Michael Osera
Department of Computer Science
Grinnell College
United States of America
osera@cs.grinnell.edu

Abstract

We explore an approach to type-directed program synthesis rooted in constraint-based type inference techniques. By doing this, we aim to more efficiently synthesize polymorphic code while also tackling advanced typing features such as GADTs that build upon polymorphism. Along the way, we also present an implementation of these techniques in Scythe, a prototype live, type-directed programming tool for the Haskell programming language and reflect on our initial experience with the tool.

CCS Concepts • **Software and its engineering → Semantics**; **Automatic programming**; • **Theory of computation** → *Logic and verification.*

Keywords Functional Programming, Program Synthesis, Type Inference, Type Theory

ACM Reference Format:
Peter-Michael Osera. 2019. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '19), August 18, 2019, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3331554.3342608

1 Introduction

Functional programmers frequently comment how richly-typed functional programs just "write themselves".[1] For example, consider writing down a function that obeys the type:

```
f :: a -> Maybe a -> a
```

Because the return type of the function, `a`, is polymorphic we can only produce a value from two sources:

1. The first argument to the function (of type `a`).
2. The result of pattern matching on the second argument (in the `Just` case, its argument will have type *a*).

These restrictions highly constrain the set of valid programs that will typecheck, giving the impression that the function writes itself as long as the programmer can navigate the type system appropriately. To do this, they must systematically check the context to see what program elements are relevant to their final goal and try to put them together into a complete program. However, such navigation is usually mechanical and tedious in nature. It would be preferable if a tool automated some or all of this *type-directed development* process.

Such luxuries are part of the promise of type-directed program synthesis tools. Program synthesis is the automatic generation of programs from specification. In this particular case, the specification of our program is its rich type coupled with auxiliary information, *e.g.*, concrete examples of intended behavior.

1.1 From Typechecking to Program Synthesis

Type-directed synthesis techniques search the space of possible programs primarily through a reinterpretation of the programming language's type system [6, 19, 22]. Traditionally, type systems are specified by defining a *relation* between a context, expression, and type, *e.g.*, $\Gamma \vdash e : \tau$ declares that e has type τ under context Γ. From this specification, we would like to extract a typechecking algorithm for the language. However, because relations do not distinguish between inputs and outputs, it is sometimes not clear how to extract such an algorithm. Bi-directional typechecking systems [21] alleviate these concerns by making it explicit which components of the system are inputs and outputs of the system, typically by distinguishing the cases where we *check* that a term has a type from the cases where we *infer* that a term has a type. In the former case, the type acts as an *input* to the system where in the latter case, it is an *output*.

In both cases, the term being typechecked serves as an input to the typechecking algorithm. However, with type-directed program synthesis, we instead view the term as an output and the type as an input. For example, consider the standard function application rule found in most type systems:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

[1] Once you get over the complexity of the types!

TyDe '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6815-5/19/08…$15.00
https://doi.org/10.1145/3331554.3342608

While we can observe that the function application should be typed at the result type of the function e_1, it isn't clear which parts of the relations are inputs and outputs and the order in which the checks ought to be carried out. A bidirectional interpretation of this rule makes the inputs and outputs explicit:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_1 \to \tau_2}$$

Here, the relation $\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2$ means that we infer that the type of e_1 is $\tau_1 \to \tau_2$. The relation $\Gamma \vdash e_2 \Leftarrow \tau_1$ means that we check that the type of e_2 is indeed τ_1. With this, the procedure for type checking a function application is clear: infer a function type $\tau_1 \to \tau_2$ for e_1 and then check that e_2 has that input type τ_1; the type of the overall application is then inferred to be the output type τ_2.

With type-directed synthesis, we turn typechecking into *term generation* by reinterpreting the inputs and outputs of the typechecking relation.

$$\frac{\Gamma \vdash \tau_1 \to \tau_2 \Rightarrow e_1 \qquad \Gamma \vdash \tau_1 \Rightarrow e_2}{\Gamma \vdash \tau_2 \Rightarrow e_1\ e_2}$$

The relation $\Gamma \vdash \tau \Rightarrow e$ asserts that whenever we have a *goal type* τ we can generate a term e of that type. Now our term generation rule for function application says that whenever we have a goal type τ_2, we can generate a function application $e_1\ e_2$ where the output of the function type agrees with the goal. We can apply this pattern to the other rules of the type system to obtain a complete term generation system for a language. We can further augment the system with type-directed example decomposition [6, 19] or richer types such as refinements [22] to obtain true program synthesis systems.

This type-theoretic interpretation of program synthesis gives us immediate insight into how to synthesize programs for languages with rich type systems. Because rich types constrain the set of possible programs dramatically, program synthesis with types can lead to superior synthesis performance. On top of this, the type-directed synthesis style directly supports type-directed programming, a hallmark of richly-typed functional programming languages.

1.2 The Perils of Polymorphism

However, in supporting rich types, in particular polymorphism, we run into a pair of problems that deserve special attention.

- **The type enumeration problem**: when dealing with polymorphic types, we must first instantiate them. However, there may be many possible instantiations of a polymorphic type. For example, consider the `map` function of type `(a -> b) -> [a] -> [b]`. Even if we know that we need to create a value of type `[Bool]`, there is nothing directly constraining the type variable `a`.
  Current systems that handle polymorphic synthesis simply enumerate and explore all the possible types that can be created from the context. However, this may lead to an excessive exploration of type combinations that do not work, *e.g.*, choosing `a` to be `Int` but not having any functions of type `Int -> Bool` available in the context. Furthermore, it may lead to repeated checking of terms that are, themselves, polymorphic, *e.g.*, the empty list `[] :: [a]` for any type `a`. We would like to develop a system that systematically searches the space of polymorphic instantiations while minimizing the work done as much as possible.
- **Reasoning about richer types**: polymorphic types are relatively easy to handle in an ad hoc fashion. However, polymorphic types form the basis for a variety of advanced type features such as generalized algebraic datatypes (GADTs) and typeclasses that are commonly used in advanced functional programming languages. Rather than developing ad hoc solutions for all these related features, it would be useful to have a single framework for tackling them all at once.

1.3 Outline

In this paper we present a solution to the problems described above: a constraint-based approach to type-directed program synthesis. Constraint-based typing is used primarily to specify type inference systems which generate constraints between types in the program and then solves those constraints to discover the types of unknown type variables. In the spirit of type-directed program synthesis, we flip the inputs and outputs of the constraint-based typing system to arrive at a synthesis system that tracks type constraints throughout the synthesis process. This embodies a new strategy for synthesis—"infer types while synthesizing"—that allows for efficient synthesis of polymorphic code while also giving us a framework to tackle GADTs and other advanced type features built on polymorphism.

In section 2, we develop a constraint-based program synthesis based on a basic constraint-based type inference system. In section 3, we extend the system to account for GADTs. In section 4, we discuss our prototype implementation of this approach in our program synthesis tool, Scythe. And finally in section 5 and section 6 we close by discussing future extensions and related work.

2 Constraint-based Program Synthesis

We first develop a constraint-based program synthesis system for a core functional programming language featuring algebraic datatypes and polymorphism. To do this, we first present a standard Hindley-Milner style type inference system for the language and then show how to re-interpret it

$$
\begin{array}{llll}
\sigma & ::= & \forall \overline{a_i}^{\,i}.\, \tau & \text{Type Schemes} \\
\tau & ::= & \alpha \mid a \mid T\ \overline{\tau_i}^{\,i} \mid \tau_1 \to \tau_2 & \text{Monotypes} \\
e & ::= & x \mid K \mid \lambda x.\, e \mid e_1\ e_2 \mid \textbf{case}\ e\ \textbf{of}\ \overline{m_j}^{\,j} & \text{Expressions} \\
m & ::= & K\ \overline{x_k}^{\,k} \to e & \text{Match Branches} \\
b & ::= & x = e & \text{Top-level Bindings} \\
 & & & \\
C & ::= & \{\overline{\tau_i \sim \tau_i'}^{\,i}\} & \text{Constraints} \\
\theta & ::= & \cdot \mid [\tau/a]\theta \mid [\tau/\alpha]\theta & \text{Type Environments} \\
\Gamma & ::= & \cdot \mid x{:}\sigma, \Gamma \mid K{:}\forall \overline{a_i}^{\,i}.\, C \Rightarrow \overline{\tau_k}^{\,k} \to T\ \overline{a_i}^{\,i}, \Gamma & \text{Contexts}
\end{array}
$$

Figure 1. Basic syntax of the object language.

as a program synthesis system, augmented with example propagation in the style of Myth [19].

Figure 1 gives the syntax of the object language. Types are composed of top-level type schemes σ which introduce universally quantified type variables. The language of monotypes τ enclosed in type schemes include type variables a, function types, and saturated algebraic datatypes $T\ \overline{\tau_j}^{\,j}$. The constraint-generation process also generates unification variables $\alpha, \beta, \gamma, \ldots$, that are eventually substituted away during the inference process. The context Γ records both the type of variables and constructors K, enforcing that all constructors are (possibly) polymorphic functions whose co-domain is their associated algebraic datatype. The co-domain is restricted to the universally quantified type variables of the declaration; in section 3, we lift this restriction to enable generalized algebraic datatypes.

The expression language is a standard typed, functional language with algebraic datatype constructors and pattern matching restricted to one-level peeling of head constructors from their arguments. Like Haskell, we treat constructors like function application and allow for partial application of constructor values. Conspicuously absent from the expression language are let-bindings. This is because we do not synthesize let-bindings (that are not top-level) during the synthesis process. As we shall see shortly, there is no type-or-example-directed way to synthesize such bindings in the general case, so we elide them from our object language.

Throughout this paper, we refer to sequences of objects either using ellipses, *e.g.*, $e_1 \ldots e_k$, or using overbar notation $\overline{e_k}^{\,k}$ where we use index variables $i, j, k, \ldots$ to represent the lengths of these sequences. We denote nested sequences (*i.e.*, sequences of sequences) using nested overbars and both subscripts and superscripts to separate the lengths. For example, $\overline{\overline{e_j}^{\,j}}^{\,i}$ refers to a sequence (of size i) of sequences (of size j) of expressions e denoted e_j^i. Finally, whenever possible we use the metavariables i, j, and k to denote the lengths of the following sequences:

- i: types, *e.g.*, $\forall \overline{a_i}^{\,i}.\, \tau$, $T\ \overline{\tau_i}^{\,i}$,
- j: examples, *e.g.*, $\overline{\{S_j \to \chi_j\}}^{\,j}$, and
- k: arguments (to constructors), *e.g.*, $K\ \overline{e_k}^{\,k}$.

$$
\boxed{\Gamma \vdash e : \tau \Rightarrow C}
$$

$$
\textsc{infer-var}\ \frac{\text{fresh}\ \overline{\alpha_i}^{\,i} \qquad x{:}\forall \overline{a_i}^{\,i}.\, \tau \in \Gamma}{\Gamma \vdash x : \overline{[\alpha_i/a_i]}^{\,i}\, \tau \Rightarrow \cdot}
$$

$$
\textsc{infer-app}\ \frac{\text{fresh}\ \alpha \qquad \Gamma \vdash e_1 : \tau_1 \Rightarrow C_1 \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow C_2}{\Gamma \vdash e_1\ e_2 : \alpha \Rightarrow C_1 \cup C_2 \cup \{\tau_1 \sim \tau_2 \to \alpha\}}
$$

$$
\textsc{infer-case}\ \frac{\text{fresh}\ \overline{\alpha_i}^{\,i}, \beta \qquad \Gamma \vdash e : \tau \Rightarrow C \qquad C' = C \cup \{\tau \sim T\ \overline{\alpha_i}^{\,i}\} \qquad \overline{\Gamma \vdash \overline{\alpha_i}^{\,i}; b_j : \tau_j \Rightarrow C_j}^{\,j}}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \overline{b_j}^{\,j} : \beta \Rightarrow C' \cup \overline{C_j}^{\,j} \cup \{\overline{\beta \sim \tau_j}^{\,j}\}}
$$

$$
\textsc{infer-lam}\ \frac{\text{fresh}\ \alpha \qquad x{:}\alpha, \Gamma \vdash e : \tau \Rightarrow C}{\Gamma \vdash \lambda x.\, e : \alpha \to \tau \Rightarrow C}
$$

$$
\boxed{\Gamma \vdash \overline{\alpha_i}^{\,i}; m : \tau \Rightarrow C}
$$

$$
\textsc{infer-match}\ \frac{K{:}\forall \overline{a_i}^{\,i}.\, \overline{\tau_k}^{\,k} \to T\ \overline{a_i}^{\,i} \in \Gamma \qquad \overline{x_k{:}\overline{[\alpha_i/a_i]}^{\,i}\, \tau_k}^{\,k}, \Gamma \vdash e : \tau \Rightarrow C}{\Gamma \vdash \overline{\alpha_i}^{\,i}; K\ \overline{x_k}^{\,k} \to e : \tau \Rightarrow C}
$$

$$
\boxed{\Gamma \vdash b : \sigma}
$$

$$
\textsc{infer-bind}\ \frac{\Gamma \vdash e : \tau \Rightarrow C \qquad \text{unify}(C) = \theta \qquad \overline{\alpha_i}^{\,i} \notin \text{fuvs}(\theta) \qquad \text{fresh}\ \overline{a_i}^{\,i}}{\Gamma \vdash f = e : \forall \overline{a_i}^{\,i}.\, \overline{[a_i/\alpha_i]}^{\,i}\, \theta\tau}
$$

Figure 2. Type inference system.

2.1 Type Inference

Figure 2 gives the type inference rules for the system. As is standard, our constraint-based type inference system operates in two phases: type-and-constraint generation and constraint solving. The primary judgment of the system $\Gamma \vdash e : \tau \Rightarrow C$ infers that expression e has type τ under context Γ and generates constraints C. Constraints in this system $\tau_1 \sim \tau_2$ assert equalities between types, in particular,

giving concrete types to *unification variables* α that are generated for unknown types during the inference process. For example, in the case of an un-annotated lambda (INFER-LAM), we do not immediately know the type of the argument to the lambda x. Thus, we create a fresh unification variable α and use that variable as the type of the argument. In inferring the type of the body of the lambda, we will generate constraints C that refine the type of α.

In addition to assigning unification variables to types, constraints also serve the purpose of asserting expected equalities between types of sub-expressions. When inferring the type of a function application (INFER-APP), we assert that the type of the head expression is indeed a function type and the argument expression's type is the domain of that function type ($\tau_1 \sim \tau_2 \to \alpha$). When inferring the type of a case scrutinee (INFER-CASE), we assert that the type of the scrutinee is indeed an algebraic datatype with fresh unification variables in place of its arguments ($\tau \sim T\ \overline{\alpha_i}^i$). And finally, we add constraints when inferring the overall type of a case expression stating that all of the branches have the same type ($\overline{\beta \sim \tau_j}^j$).

Polymorphic types interact with the inference system in two ways. The first is *let-generalization* which occurs at top-level bindings $f = e$. After generating constraints C for the body of the binding, we solve them using a standard unification algorithm to find a type substitution θ for each of the unification variables found in C (INFER-BIND). Those unification variables with no substituting types ($\forall \overline{\alpha_i}^i \notin \text{fuvs}(\theta)$) become universal type variables in the final type scheme of f. The second way lies in inferring the monotypes we use to instantiate the type schemes of variables. Rather than requiring that the user provides the instantiation via a type application form or guessing the instantiation upfront, we instantiate a type scheme with fresh unification variables $\overline{\alpha_i}^i$ that will be later constrained based on how the the variable x is used by the program (INFER-VAR).

2.2 Constraint-based Synthesis

Figure 3 gives the syntactic extensions to the core language to support type-and-example-directed program synthesis in the style of MYTH [6, 19]. We extend the language with example values χ that the user provides as additional specification to the synthesizer. Intuitively, example values are specifications of values at a particular type that can be decomposed into specifications for those values' components. For example, each of the values $\overline{v_i}$ of a saturated constructor example $K\ \overline{v_i}$ can become examples for synthesizing each of K's arguments.

However, it is not immediately clear how to decompose values at function type, *i.e.*, lambdas, in a similar way. Like MYTH, we introduce *input-output pairs* $v \Rightarrow \chi$ as a distinguished example value at function type. Such examples are easily decomposed and align with our intuition that we would like to specify the behavior of a function through a collection of input-output examples (either realized as test cases or documentation).

The synthesis system takes as input a context Γ, a goal type τ, and a set of examples, and produces a program of that type that agrees with the set of examples. Ultimately, the system decomposes the examples, assigning sub-values to generated binders—from lambdas and case expressions—as the synthesizer generates them. These are recorded alongside the examples that generated them, leading to the notion of an *example world* W as a pair of a (value) environment S and a *example goal* χ for that world. When checking that a program agrees with these examples, we evaluate the program closed with each example world's environment and check that the resulting value is consistent with that world's example goal. We write the application of an environment S to an expression e using juxtaposition: Se.

Figure 4 describes the complete synthesis system over the language. In the spirit of bidirectional typechecking [21], we divide type-directed synthesis into two processes which are realized as a pair of judgments in the system:

1. *Refinement*, $\Gamma \vdash \tau \rhd W \Rightarrow e$, decomposes a set of input examples according to their types.
2. *Generation*, $\Gamma \vdash \tau \Rightarrow e$ enumerates terms in a type-directed fashion in the absence of examples.

The separation is, in fact, a division of the syntax of the language into *introduction* forms and *elimination* forms for refinement and generation, respectively. From a type inference perspective, type information flows *into* introduction forms (*i.e.*, we check against their expected types based on their shape) and *out of* elimination forms (*i.e.*, we infer their types based on their sub-components). From a synthesis perspective, this means we can use the shape of an introduction form to determine how to decompose examples whereas we have no such affordances in the general case for elimination forms. Thus, we must resort to raw term enumeration for those forms.

As a final note, the system as described does not support recursive functions (note that in Figure 4, REFINE-LAM does not bind anything for the function itself) in order to make clear the details of constraint propagation in the synthesis process. Recursion can be easily integrated into the system in the same manner as MYTH by treating the set of input-output examples for a function as a *partial function* and using that function value as a binding for the function-being-synthesized. This requires that the example set is trace complete so that any call to the function has a known value [19] and is how the SCYTHE system discussed in section 4 implements recursion.

2.3 Refinement

The non-polymorphic refinement rules are largely identical to those found in MYTH. The function of each rule is to

$$
\begin{array}{llll}
v & ::= & c \mid \lambda x.\, e \mid K\ \overline{v_i}^{\,i} & \text{Values} \\
\chi & ::= & c \mid K\ \overline{\chi_i}^{\,i} \mid v \Rightarrow \chi & \text{Examples} \\
X & ::= & \forall \overline{c_i{:}a_i}^{\,i}.\ \overline{\chi_j}^{\,j} & \text{Top-level Examples} \\
S & ::= & \overline{[v_i/x_i]}^{\,i} & \text{Environments} \\
W & ::= & \overline{[S_j \mapsto \chi_j]}^{\,j} & \text{Example Worlds} \\
\Gamma & ::= & \cdots \mid c{:}a, \Gamma & \text{Contexts}
\end{array}
$$

Figure 3. Syntactic extensions for type-and-example-directed program synthesis.

describe how to decompose a set of example worlds according to the goal type's example form. For lambdas (REFINE-LAM), we decompose a collection of worlds containing input-output examples by assigning the lambda's binder x the input and synthesizing the body of the lambda with the output as the goal. For constructors (REFINE-DATA), if all the examples share the same head K, then we can safely synthesize that constructor and divide up the examples' arguments into examples for each of that constructor's arguments. For example, suppose that we have a constructor $K : \mathsf{Int} \to \mathsf{Bool} \to \mathsf{Int} \to T$ with example worlds $W = w_1, w_2, w_3$:

$$
\begin{aligned}
w_1 &= S_1 \mapsto K\ 0\ \mathsf{True}\ 1 \\
w_2 &= S_2 \mapsto K\ 2\ \mathsf{False}\ 3 \\
w_3 &= S_3 \mapsto K\ 4\ \mathsf{True}\ 5
\end{aligned}
$$

By REFINE-DATA, we will synthesize $K\ \blacksquare_1\ \blacksquare_2\ \blacksquare_3$ with three synthesis sub-goals for each of K's arguments. Each of the sub-goals contains example worlds W_1, W_2, and W_3 respectively:

$$
\begin{aligned}
W_1 &= S_1 \mapsto 0, S_2 \mapsto 2, S_3 \mapsto 4 \\
W_2 &= S_1 \mapsto \mathsf{True}, S_2 \mapsto \mathsf{False}, S_3 \mapsto \mathsf{True} \\
W_3 &= S_1 \mapsto 1, S_2 \mapsto 3, S_3 \mapsto 5
\end{aligned}
$$

Notably with refinement, the fully specified goal type is required as input to the procedure, so there is no need to reason about type constraints at this point in the system[2].

Unlike other type-directed forms, case-expressions do not immediately imply a particular type. Indeed the type of a case-expression is exactly the type shared by its match bodies. Nevertheless, we can refine examples through case expressions as described by the REFINE-CASE rule:

1. Generate a scrutinee expression e at some concrete datatype $T\ \tau_j$.
2. For each constructor K of T, create a match for that constructor using the example worlds $\{\overline{S_j \mapsto \chi_j}^{\,j}\} \subseteq W$ where the scrutinee would evaluate to a value with head constructor K (written using the *filtering operator* $W|_e^K$ defined in Figure 5).

When generating a scrutinee, we use the term-generation judgment $C; \Gamma \vdash \tau \Rightarrow e; C'$ which produces a set of type constraints C' under which e typechecks at type τ. However, because we assume the concrete datatype of type $T\ \overline{\tau_i}^{\,i}$ upfront, we know that the type does not contain any unification variables and thus we can ignore C' for now. In theory this means that an implementation of the system must explore all possible combinations of datatypes to arguments which is the exact problem we are trying to solve with constraint-based synthesis! But in practice, we heavily restrict the kinds of expressions that appear as scrutinees already, *e.g.*, to direct arguments of the synthesized function, so such type search is tolerable in practice.

The helper judgment $\Gamma \vdash e; \overline{\tau_i}^{\,i}; K; \tau \triangleright W \Rightarrow m$ is responsible for synthesizing match branches from the scrutinee and candidate constructor. Its sole rule (REFINE-MATCH) extracts and binds the evaluated-to constructor values' arguments in the filtered example worlds and continues synthesis of the match's body.

Finally, we connect refinement and term generation with the REFINE-GUESS rule which generates a term e and then checks to see if, for each example world $S \mapsto \chi$, that the closed term Se is equivalent to the goal example χ according to the standard evaluation rules of the language ($e \equiv_\beta v$ whenever $e \longrightarrow^* v$). The only non-standard case that can arise is the comparison of a lambda (a v) to an input-output example (a χ). We thus define:

$$\lambda x{:}\tau.\, e \equiv_\beta v \Rightarrow \chi \text{ iff } (\lambda x{:}\tau.\, e)\ v \equiv_\beta \chi.$$

That is, we run the input of the input-output pair through the lambda and check to see the resulting value is equivalent to the output of the pair.

Polymorphism in Refinement We must consider polymorphic types at two points within refinement. First, unlike the presentations of the case rule in prior work [19], constructors now have (potentially) polymorphic types. However, because the type $T\ \overline{\tau_i}^{\,i}$ is fully concrete, we can immediately instantiate the constructor's polymorphic type ($\theta = [\overline{\tau_i/a_i}^{\,i}]$) and use that type substitution on each of the types of the arguments of the constructor.

[2]Note that even with goal types elided, the examples make inference of the goal type trivial due to canonicity of types. However, we may want to consider type-directed synthesis in the absence of any examples which we discuss in more detail in section 5.

$$\boxed{\Gamma \vdash \tau \rhd W \Rightarrow e}$$

REFINE-GUESS
$$\frac{C;\Gamma \vdash \tau \Rightarrow e;C' \qquad \overline{S_j e \equiv_\beta \chi_j}^{\,j} \qquad \text{consistent}(C')}{\Gamma \vdash \tau \rhd \{\overline{S_j \mapsto \chi_j}^{\,j}\} \Rightarrow e}$$

REFINE-LAM
$$\frac{\text{fresh}\, x \qquad x{:}\tau_1, \Gamma \vdash \tau_2 \rhd \{\overline{[v_j/x]S_j \mapsto \chi_j}^{\,j}\} \Rightarrow e}{\Gamma \vdash \tau_1 \to \tau_2 \rhd \{\overline{S_j \mapsto v_j \Rightarrow \chi_j}^{\,j}\} \Rightarrow \lambda x.\, e}$$

REFINE-DATA
$$\frac{\dfrac{K{:}\forall \overline{a_i}^{\,i}.\, \overline{\tau_k}^{\,k} \to T\ \overline{a_i}^{\,i} \in \Gamma \qquad \theta = [\overline{\tau_i/a_i}^{\,i}]}{W_k = \{\overline{S_j \mapsto \chi_k^j}^{\,j}\}}^{\,k} \qquad \overline{\Gamma \vdash \theta\tau_k \rhd W_k \Rightarrow e_k}^{\,k}}{\Gamma \vdash T\ \overline{\tau_i}^{\,i} \rhd \{S_j \mapsto K\ \overline{\chi_k^j}^{\,k}\} \Rightarrow K\ \overline{e_k}^{\,k}}$$

REFINE-CASE
$$\frac{C;\Gamma \vdash T\ \overline{\tau_i}^{\,i} \Rightarrow e;C' \qquad \overline{K_j{:}\forall \overline{a_i}^{\,i}.\, \overline{\tau_k^j}^{\,k} \to T\ \overline{a_i}^{\,i} \in \Gamma}^{\,j} \qquad \overline{\Gamma \vdash e;\overline{\tau_i}^{\,i};K_j;\tau \rhd W \Rightarrow m_j}^{\,j}}{\Gamma \vdash \tau \rhd W \Rightarrow \textbf{case}\ e\ \textbf{of}\ \overline{m_j}^{\,j}}$$

$$\boxed{\Gamma \vdash e;\overline{\tau_i}^{\,i};K;\tau \rhd W \Rightarrow m}$$

REFINE-MATCH
$$\frac{\dfrac{K{:}\forall \overline{a_i}^{\,i}.\, \overline{\tau_k}^{\,k} \to T\ \overline{a_i}^{\,i} \in \Gamma \qquad W_e^K = \overline{\{S_j \mapsto \chi_j\}}^{\,j} \qquad \theta = [\overline{\tau_i/a_i}]^i \qquad S_j e \longrightarrow^* K\ \overline{v_k^j}^{\,k} \qquad \text{fresh}\, \overline{x_k}^{\,k}}{\overline{x_k{:}\theta\tau_k}^{\,k}, \Gamma \vdash \tau \rhd \{\overline{[v_k^j/x_k]\ S_j \mapsto \chi_j}^{\,k}\} \Rightarrow e'}^{\,j}}{\Gamma \vdash e;\overline{\tau_i}^{\,i};K;\tau \rhd W \Rightarrow K\ \overline{x_k}^{\,k} \to e'}$$

$$\boxed{\Gamma \vdash \sigma \rhd W \Rightarrow e}$$

REFINE-POLY
$$\frac{\Gamma \vdash \tau \rhd \{\overline{S_j \mapsto \chi_j}^{\,j}\} \Rightarrow e}{\Gamma \vdash \forall \overline{a_i}^{\,i}.\, \tau \rhd \{\overline{S_j \mapsto \forall \overline{c_k{:}a_k}^{\,k}.\, \chi_j}^{\,j}\} \Rightarrow e}$$

$$\boxed{C;\Gamma \vdash \tau \Rightarrow e;C'}$$

GEN-VAR
$$\frac{\text{fresh}\, \overline{\alpha_i}^{\,i} \qquad x{:}\forall \overline{a_i}^{\,i}.\, C' \Rightarrow \tau' \in \Gamma \qquad \theta = [\overline{\alpha_i/a_i}]^i \qquad C'' = C \cup \theta C' \cup \{\tau \sim \theta\tau'\} \qquad \text{consistent}(C'')}{C;\Gamma \vdash \tau \Rightarrow x;C''}$$

GEN-APP
$$\frac{\text{fresh}\, \alpha \qquad C;\Gamma \vdash \alpha \to \tau \Rightarrow e_1;C_1 \qquad C_1;\Gamma \vdash \alpha \Rightarrow e_2;C_2}{C;\Gamma \vdash \tau \Rightarrow e_1\ e_2;C_2}$$

GEN-UNIFY
$$\frac{C \vDash \tau_1 \sim \tau_2 \qquad C;\Gamma \vdash \tau_2 \Rightarrow e;C'}{C;\Gamma \vdash \tau_1 \Rightarrow e;C'}$$

Figure 4. Type-and-example program synthesis rules.

$$e \equiv_\beta v \text{ iff } e \longrightarrow^* v$$
$$\lambda x{:}\tau.\, e \equiv_\beta v \Rightarrow \chi \text{ iff } (\lambda x{:}\tau.\, e)\, v \equiv_\beta \chi.$$
$$W|_e^K = \{S \to \chi \in W \mid Se \longrightarrow^* K\ v_1\ \cdots\ v_k\}$$
$$\text{unify}(C) = \theta \qquad \text{(standard unification algorithm)}$$
$$C \vDash C' \text{ iff } \forall \tau \sim \tau' \in C'.\, \theta\tau = \theta\tau' \text{whereunify}(C) = \theta.$$
$$\text{consistent}(C) \text{ iff } \exists \theta.\, \text{unify}(C) = \theta$$

Figure 5. Auxiliary definitions for the synthesis system.

Secondly, top-level bindings are at type schemes σ, so we must provide examples at polymorphic type for them. However, there is no type-directed syntactic form for polymorphic types! Furthermore, even an explicit term-level type abstraction, *e.g.*, $\Lambda a.\, \chi$, does not provide much help! This is because the type abstraction does not introduce *values* of the type variable a, and we need such values to write down complete examples.

We therefore introduce a new top-level example value form X in Figure 3, the *polymorphic example* $\forall \overline{c_i{:}a_i}^{\,i}.\, \overline{\chi_j}^{\,j}$, which introduces a set of *polymorphic constants* c_i at type variables a_i. This example form was first introduced in Osera's thesis [17] in the context of synthesizing within System F, and we have adapted it to this Haskell-like setting. For example, we might specify for a goal type of $\forall a.\, a \to$ **Maybe** $a \to a$:

$$\forall c_1{:}a, c_2{:}a.\, c_1 \Rightarrow \textbf{Nothing} \Rightarrow c_1,$$
$$c_1 \Rightarrow \textbf{Just}\ c_2 \Rightarrow c_2.$$

In these two examples, we introduce polymorphic constants c_1 and c_2 of type a and use them throughout the input-output examples for the standard `fromMaybe` function.[3]

We refine polymorphic examples introduced at top-level bindings (REFINE-POLY) by simply stripping the top-level forall and synthesizing at the contained sub-example value. The binding information for the polymorphic constant is only useful for typechecking the examples themselves; they are not necessary for the synthesis process as the example values are simply carried around as values bound to variables in an example world's environment.

2.4 Generation

Like refinement, term generation takes as input a goal type. However, during the course of generation, we may need to instantiate types schemes when generating variables. Rather than committing to a choice of a complete instantiation of a type scheme upfront (which requires us to systematically

[3] The astute reader ought to notice that these examples are virtually the definition of `fromMaybe` already! We reflect on this fact in more detail in section 4.

search the space of possible types), we infer the instantiation as we generate the complete term. If type-and-example-directed program synthesis captures the idea of "evaluating *during* enumeration" [19], then our approach of integrating type inference into the term generation adds the idea of "inferring *during* enumeration".

To do so, we "invert" the appropriate rules from our type inference system (Figure 2) to create a term generation system that infers types during the generation process. This judgment, written $C; \Gamma \vdash \tau \Rightarrow e; C'$, takes a collection of constraints C, a context Γ, and type τ as input as produces a program e and an updated constraint set C' as output. In effect, we thread a set of type constraints throughout the generation process which makes concrete the idea that a choice of a component in one part of an expression might influence choices in the rest of the expression. It is worthwhile to note that our rules are not an exact inversion of the type inference system defined in Figure 2 because the generation process is given a concrete goal type to work with as input.

While refinement handles the introduction forms of the language, generation concerns the elimination forms, of which there are only variables and function application. With variable generation (GEN-VAR), we speculatively choose a variable to generate and then create a new constraint recording that the goal type must be equivalent to the type scheme of the variable instantiated with fresh unification variables ($\tau \sim [\overline{\alpha_i / a_i}^{\,i}]\tau'$). And with function application generation (GEN-APP), we first speculatively generate a function under the constraint that its domain matches our eventual argument type and the co-domain matches our goal type. We then generate function arguments under the constraints accrued from generating the function.

While the initial goal type for generation is a concrete type, we will quickly introduce unknown types, *i.e.*, unification variables, through the process. We eliminate unification variables through unification of the constraints during the generation process (GEN-UNIFY). If our goal type is some τ_1 (presumably a unification variable) and our current constraint set allows us to conclude that it is equivalent to τ_2 ($C \vDash \tau_1 \sim \tau_2$) then we can synthesize at this type τ instead.

As we generate constraints, we need to take care that these constraints are consistent. An inconsistent set of constraints asserts that in-equal types are equal, *e.g.*, **Int** $\sim$ **Bool**. If at any point we arrive at a set of inconsistent constraints, we know that the currently generated term is not well-typed and should be rejected. In our formal system, this amounts to using the consistent helper relation to check that whenever we output a constraint set from a rule that it is indeed consistent. consistent(C) simply asserts that a unifier (*i.e.*, type substitution) exists for C as described in Figure 5.

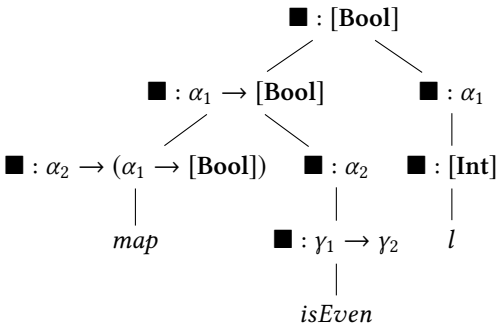

Figure 6. Example generation derivation of *map*.

For example, consider generating a program of type [**Bool**] under the context:

$$\begin{aligned}\Gamma = \; & \mathit{map} : \forall ab.\, (a \to b) \to [a] \to [b], \\ & \mathit{isEven} : \mathbf{Int} \to \mathbf{Bool}, \\ & l : [\mathbf{Int}]\end{aligned}$$

with goal type [**Bool**]. Figure 6 graphically shows one potential generation of a program of this type.

Starting at the root, we first apply the GEN-APP rule which generates a fresh unification variable α_1 and two generation sub-goals at types $\alpha_1 \to [\mathbf{Bool}]$ and α_1. We apply GEN-APP again at the first sub-goal which generates another unification variable α_2 and two more sub-goals at types $\alpha_2 \to (\alpha_1 \to [\mathbf{Bool}])$ and α_2.

At the first of these new sub-goals, we can apply GEN-VAR and choose the *map* variable. This generates the constraint

$$\alpha_2 \to \alpha_1 \to [\mathbf{Bool}] \sim (\gamma_1 \to \gamma_2) \to [\gamma_1] \to [\gamma_2]$$

for fresh unification variables γ_1 and γ_2. Now we can return to the sub-goal with goal type α_2 and apply GEN-UNIFY to continue generating at type $\gamma_1 \to \gamma_2$ since our constraint implies that $\alpha_2 \sim \gamma_1 \to \gamma_2$. This allows us to apply GEN-VAR and choose the *isEven* variable, adding the constraint

$$\gamma_1 \to \gamma_2 \sim \mathbf{Int} \to \mathbf{Bool}.$$

Finally, we can return to the sub-goal with goal type α_1, apply GEN-UNIFY to generate at type [**Int**] (because $\alpha_1 \sim [\gamma_1] \sim [\mathbf{Int}]$), and the choose l via GEN-VAR, generating a final, trivial constraint [**Int**] $\sim$ [**Int**].

Because we are ensuring that our generated constraint sets are consistent at every step, the order in which we explore generation sub-goals doesn't matter. However, the order heavily influences how much backtracking we will need to perform because of "bad" choices of terms. For example, if rather than synthesizing *map* first, we instead try to synthesize its arguments, we may make many spurious choices if there are non-list types in the context. Thus, even though our constraint-based synthesis system allows us to consider

$$\Gamma ::= \cdots \mid K{:}\forall \overline{a_i}^{\,i}.\, C \Rightarrow \overline{\tau_k}^{\,k} \to T\ \overline{a_i}^{\,i}, \Gamma \quad \text{Contexts}$$

Figure 7. Syntactic extensions for GADT synthesis.

types in a more incremental fashion, we must still be conscious of performance during implementation. In this particular case, it makes sense to attempt to synthesize functions before their arguments since the function is more likely to prune the search space of types than synthesizing one of its arguments.

3 Synthesis with GADTs

Traditional algebraic types require that the output type of its constructors have the form $T\ \overline{a_i}^{\,i}$ where the a_is are universally quantified type variables from the type of the constructor. *Generalized algebraic data types* (GADTs) lift this restriction so that the type arguments of T may be any type, not just type variables. While a seemingly insignificant change, GADTs strike a powerful balance between power and complexity on the spectrum of rich type systems.

The quintessential example of GADTs-in-action is the tagless interpreter given below in Haskell:

```
data Exp a where
  Lit  :: a -> Exp a
  Plus :: Exp Int -> Exp Int -> Exp Int
  Eq   :: Exp Int -> Exp Int -> Exp Bool
  If   :: Exp Bool -> Exp a -> Exp a -> Exp a

eval :: Exp a -> a
eval (Lit x)       = x
eval (Plus e1 e2)  = eval e1 + eval e2
eval (Eq e1 e2)    = eval e1 == eval e2
eval (If e1 e2 e3) =
  if eval e1 then eval e2 else eval e3
```

Here, we parameterize the type of `Exp a` by the type `a` that the expression evaluates to. This allows us to give `eval` the rich type `Exp a -> a` which in turn allows us to write `eval` without any case analyses on the values it produces.

How do GADTs complicate typechecking and consequently type-directed program synthesis? For example, consider the `Plus` case of `eval`. If `Plus` was a regular algebraic datatype (say, with type `Exp -> Exp -> Exp`), the body would not typecheck because `(+)` produces a value of numeric type whereas the function expects a value of type `a`. To enable the body of `Plus` to typecheck, the typechecker must deduce that because the input is a constructor that produces an `Exp Int`, then `a` ~ `Int` and thus we must typecheck the body at type `Int`. In effect, consuming a GADT via a pattern match introduces *local type constraints* into the environment that we must consider when typechecking each branch [25].

With some minor changes, we can adapt the constraint-based framework we developed previously to accommodate GADTs. In particular, we follow the presentation of OutsideIn [25] and represent GADTs as standard polymorphic constructors coupled with constraints C on the type variables that appear in the output type of the constructor. This allows us to avoid a step of unification during type inference and synthesis to discover these constraints on our own. As an example, we can rewrite the declaration of `Exp` above in this style:

```
data Exp a where
  Lit  :: a -> Exp a
  Plus :: (a ~ Int) =>
            Exp Int -> Exp Int -> Exp a
  Eq   :: (a ~ Bool) =>
            Exp Int -> Exp Int -> Exp a
  If   :: Exp Bool -> Exp a -> Exp a -> Exp a
```

With this style, our GADT constructor definitions obey the same restriction as regular algebraic datatypes—the type arguments of the output type are type variables—but the addition of type constraints allow us greater freedom as to what types the constructors produce.

Figure 8 gives the updated synthesis rules with generalized algebraic datatypes. Because GADTs only affect datatype declarations and constructors, we only need to update the rules for refinement; raw term generation is left untouched.

We update the refinement judgment to carry around the set of constraints C gained through case analysis of GADT values. Unlike term generation, additional constraints gained through GADT analysis are scoped to their matches. Because of this, we only need to thread constraints down the refinement judgment as an input and not gather updated constraints as an output. Most rules simply pass their constraints to their sub-refinement calls as in the case of lambdas (REFINE-GADT-LAM) or to raw term generation (REFINE-GADT-GUESS). At the top-level, we begin refinement with the empty set of constraints (REFINE-GADT-BINDING).

Now when generating a case expression (REFINE-GADT-CASE), we also extract the constraints associated with each constructor C_i and add it to the constraint set associated with synthesizing that constructor's match body. Notably, we instantiate the additional constraints C_i with the type environment θ generated by unifying the type of the scrutinee with the output type of the constructors, written θC_i. Constraints are utilized during refinement when synthesizing constructors (REFINE-GADT-DATA). We ensure that we only synthesize a constructor if the current set of constraints C implies all of the (instantiated) constraints bundled with the constructor's type. Note that this stands in contrast to inconsistencies detected during case generation. If the constraints of a constructor are inconsistent with the current set of constraints when synthesizing a match alternative then the alternative is unreachable and can be elided as an optimization.

$$\boxed{C; \Gamma \vdash \tau \triangleright W \Rightarrow e}$$

REFINE-GADT-LAM

$$\frac{\text{fresh } x \qquad C; x{:}\tau_1, \Gamma \vdash \tau_2 \triangleright \{\overline{[v_j/x]S_j \mapsto \chi_j}^{\,j}\} \Rightarrow e}{C; \Gamma \vdash \tau_1 \to \tau_2 \triangleright \{\overline{S_j \mapsto v_j \Rightarrow \chi_j}^{\,j}\} \Rightarrow \lambda x.\, e}$$

REFINE-GADT-DATA

$$\frac{\begin{array}{c} K{:}\forall \overline{a_i}^{\,i}.\, C' \Rightarrow \overline{\tau_k}^{\,k} \to T\ \overline{a_i}^{\,i} \in \Gamma \\ \theta = [\overline{\tau_i/a_i}^{\,i}] \qquad C \vDash \theta C' \\ \overline{W_k = \{S_j \mapsto \chi_k^j\}}^{\,j,k} \qquad \overline{C; \Gamma \vdash \theta\tau_k \triangleright W_k \Rightarrow e_k}^{\,k} \end{array}}{C; \Gamma \vdash T\ \overline{\tau_i}^{\,i} \triangleright \{S_j \mapsto K\ \overline{\chi_k^j}^{\,k}\} \Rightarrow K\ \overline{e_k}^{\,k}}$$

REFINE-GADT-CASE

$$\frac{C; \Gamma \vdash T\ \overline{\tau_i}^{\,i} \Rightarrow e; C' \qquad \overline{K_j{:}\forall \overline{a_i}^{\,i}.\, C_j \Rightarrow \overline{\tau_k^j}^{\,k} \to T\ \overline{a_i}^{\,i} \in \Gamma}^{\,j} \qquad \overline{C; \Gamma \vdash e; \overline{\tau_i}^{\,i}; K_j; \tau \triangleright W \Rightarrow m_j}^{\,j}}{C; \Gamma \vdash \tau \triangleright W \Rightarrow \textbf{case } e \textbf{ of } \overline{m_j}^{\,j}}$$

REFINE-GADT-GUESS

$$\frac{C; \Gamma \vdash \tau \Rightarrow e; C' \qquad \overline{S_j e \equiv_\beta \chi_j}^{\,j} \qquad \text{consistent}(C')}{C; \Gamma \vdash \tau \triangleright \{\overline{S_j \mapsto \chi_j}^{\,j}\} \Rightarrow e}$$

REFINE-GADT-UNIFY

$$\frac{C \vDash a \sim \tau \qquad C; \Gamma \vdash \tau \triangleright W \Rightarrow e}{C; \Gamma \vdash a \triangleright W \Rightarrow e}$$

$$\boxed{C; \Gamma \vdash e; \overline{\tau_i}^{\,i}; K; \tau \triangleright W \Rightarrow m}$$

REFINE-GADT-MATCH

$$\frac{\begin{array}{c} K{:}\forall \overline{a_i}^{\,i}.\, C' \Rightarrow \overline{\tau_k}^{\,k} \to T\ \overline{a_i}^{\,i} \in \Gamma \qquad W|_e^K = \overline{\{S_j \mapsto \chi_j\}}^{\,j} \\ \theta = \overline{[\tau_i/a_i]}^{\,i} \qquad S_j e \longrightarrow^* K\ \overline{v_k^j}^{\,k} \qquad \text{fresh } \overline{x_k}^{\,k} \\ C \cup \theta C'; \overline{x_k{:}\theta\tau_k}^{\,k}, \Gamma \vdash \tau \triangleright \{\overline{[v_k^j/x_k]\ S_j \mapsto \chi_j}^{\,k}\}^{\,j} \Rightarrow e' \end{array}}{C; \Gamma \vdash e; \overline{\tau_i}^{\,i}; K; \tau \triangleright W \Rightarrow K\ \overline{x_k}^{\,k} \to e'}$$

$$\boxed{\Gamma \vdash \sigma \triangleright X \Rightarrow b}$$

REFINE-GADT-BINDING

$$\frac{\text{fresh } x \qquad \text{ftvs}(\tau) \subseteq \overline{a_i}^{\,i} \qquad \cdot; \overline{c_k{:}a_k}^{\,k}, \Gamma \vdash \tau \triangleright \{\cdot \mapsto \overline{\chi_j}^{\,j}\} \Rightarrow e}{\Gamma \vdash \forall \overline{a_i}^{\,i}.\, \tau \triangleright \forall \overline{c_k{:}a_k}^{\,k}.\, \overline{\chi_j}^{\,j} \Rightarrow x = e}$$

Figure 8. Extensions to synthesis system for GADTs.

Finally, like with term generation, we need to use the constraints in refinement to refine the goal type when appropriate. The unification rule for refinement (REFINE-GADT-UNIFY) operates identically to its generation counterpart but at type variables: at a type variable a, we can use a's assigned type as recorded by C instead.

As an example of the system in action, consider how constraints are generated and utilized while synthesizing part of the `eval` function introduced at the beginning of this section. After an initial application of REFINE-GADT-BINDING and REFINE-GADT-LAM, we arrive at the following refinement state:

$$\begin{aligned} eval\ e = \blacksquare :: a \triangleright W \\ \Gamma = eval : \forall a.\, \textbf{Exp}\ a \to a, e : \textbf{Exp}\ a \\ C = \cdot. \end{aligned}$$

At this point, we can apply the REFINE-GADT-CASE rule to synthesize a case analysis on e. Let's just focus on the *Plus* constructor which has type:

$$\forall b.\, (b \sim \textbf{Int}) \Rightarrow \textbf{Exp Int} \to \textbf{Exp Int} \to \textbf{Exp}\ b.$$

(We α-rename the constructor's bound type variable to b in order to distinguish it from the type variable from the overall type of the function a). Unifying the return type of *Plus* with e yields the type substitution $\theta = [a/b]$ which then yields the final constraint $a \sim \textbf{Int}$ which is added to the set of constraints when synthesizing the body of *Plus*'s match at goal type a. This constraint is then passed to term generation which is then used by GEN-UNIFY to change the type of the goal from a to **Int** which then justifies synthesizing an application of the (+) function. When we go to generate recursive function calls to *eval*, the constraint is also utilized when instantiating the polymorphic type of *eval* to **Exp** $a \to a$ with $a \sim \textbf{Int}$.

3.1 Metatheory

There are two properties that we ought to consider of the synthesis system we have developed so far:

- **Soundness** states that synthesized programs are consistent with their specification.
- **Completeness** states that we are able to synthesize all programs that meet a particular specification.

Here, we briefly discuss these two properties. A summary of the complete system as well as detailed proofs can be found in the extended version of this paper [18].

Soundness The kinds of specification we consider in this system are *types* and *examples*. Thus, we can consider soundness with respect to each:[4]

Lemma 3.1 (Type Soundness). *If* $C; \Gamma \vdash \tau \triangleright W \Rightarrow e$ *then* $C; \Gamma \vdash e : \tau$.

Proof Sketch. By induction on the synthesis derivation. Intuitively, we synthesize terms in a type-directed manner (the synthesis rules are derived directly from the typing rules of the system), so we expect these terms to be well-typed. □

Lemma 3.2 (Example Soundness). *If* $C; \Gamma \vdash \tau \triangleright W \Rightarrow e$ *then for all* $S \mapsto \chi \in W$, $Se \equiv_\beta \chi$.

[4]Note that, for brevity's sake, we only state the relevant properties for expression refinement. Similar statements must be made for each of the remaining judgments of the system—match synthesis and term generation.

Proof Sketch. By induction on the synthesis derivation. During refinement, we decompose examples in such a way that example satisfaction of sub-components results in example satisfaction of the whole program. During term generation, the REFINE-GADT-GUESS rule ensures example satisfaction directly for generated terms. □

Completeness This property states that our synthesis system is capable of generating all possible programs given a fixed goal type and set of examples.

Lemma 3.3 (Completeness). *If* $C; \Gamma \vdash e : \tau$ *and* $\Gamma \vdash W : \tau$ *then* $C; \Gamma \vdash \tau \triangleright W \Longrightarrow e$.

However, our system does not obey this property in the name of efficiency! For example, note that GEN-APP does not allow for the generation of case expressions in the position of arguments.

But such cases are not terribly interesting with respect to completeness; a case nested in an argument position of an application could instead be "bubbled up" to encase the entire application (at the cost of code redundancy). Are there more interesting programs that the system is incapable of synthesizing? We leave this exploration to future work.

4 Integrating Constraint-based Synthesis in Scythe

We have implemented constraint-based synthesis for polymorphic types as described in section 2 in an in-development program synthesis tool called SCYTHE. SCYTHE supports *live, type-directed development* for the Haskell programming language by extending prior work on GHC for typed holes [7]. The typed holes facility of GHC allows users to annotate their program with holes. During typechecking, GHC will report the inferred type of each hole and suggest possible variables from the context to fill these holes. The search process currently in GHC is fairly naïve, only suggesting individual components whose type matches the goal. SCYTHE provides a far more in-depth search of the space of possible programs using the program synthesis techniques described in this work.

To do this, SCYTHE leverages the unmodified GHC API to interact with existing Haskell code with a high degree of compatibility. It uses GHC to load external libraries and parse, typecheck, interpret, and manipulate Haskell code. On the surface, this seems difficult to do since the type-and-example-directed program synthesis process interweaves a number of compilation processes, in particular, typechecking and partial evaluation. However with some engineering techniques and compromises, we can adopt off-the-shelf compiler APIs to provide this rich kind of editing experience, similar to the efforts of Merlin for OCaml [2].

As an example of the system in operation, here is the declaration of the `fromMaybe` example from subsection 2.3 as realized in the prototype version of SCYTHE:

```
--
-- | {@
--     fromMaybe :: a -> Maybe a -> a
--     fromMaybe a1 Nothing   = a1
--     fromMaybe a1 (Just a2) = a2
--     @@
--     ctx=(Just, Nothing)
--   @}
fromMaybe :: a -> Maybe a -> a
fromMaybe s1 m1 = _
```

Scythe looks for special `{@...@}` sections in Haddock comments which denote SCYTHE example blocks. These example blocks contain type-and-example information as well as options to control the synthesizer such as the `ctx` option that allows the user to specify the components to consider during synthesis. Note that like Haskell, SCYTHE infers the universal quantifier for polymorphic example values; all variables of the form `ak` where `a` is a type variable introduced by the function signature and `k` is a natural number are considered to be polymorphic constants of type `a`.

After finding the hole, SCYTHE offers the following single suggestion to complete it:

```
> (ok) case m1 of
>        Nothing -> s1
>        Just a1 -> a1
```

4.1 Experience with Polymorphic Synthesis

To date, we have performed a preliminary investigation of how type-and-example-driven polymorphic synthesis performs within SCYTHE over small toy functions. We reflect on our experiences and lessons learned so far from this investigation that are motivating our future work in this area.

Difficulties with Examples Polymorphic example values give us great flexibility in how we specify goals of polymorphic type. However, as discussed in subsection 2.3, a collection of polymorphic example values can quickly become the actual function definition itself! This demonstrates the power of combining polymorphism with example-based reasoning, but it makes the story for a tool like SCYTHE less compelling at first glance.

This is certainly true when we are forced to give many examples in order to appease the synthesizer. Early synthesizers such as MYTH had the problematic requirement of *trace completeness* that enough examples are provided so that the synthesizer can carry out execution of any recursive function call made during the synthesis process without having the entire function formed.

However, SCYTHE lifts this restriction by not immediately rejecting candidate programs that fail due to a lack of examples. A detailed discussion of this feature is beyond the scope of this paper, but the importance of this fact is that SCYTHE can produce meaningful output with little-to-no examples! As a result, polymorphic example values become

```
-- | {@
--       stutter :: [a] -> [a]
--       stutter [a1, a2] = [a1, a1, a2, a2]
--       @@
--       recArg=0
--   @}
stutter :: [a] -> [a]
stutter l = _

> (ok) case l1 of
>       [] -> l1
>       (:) a1 l2 -> (:) a1 ((:) a1 (stutter l2))
-- | {@
--       append :: [a] -> [a] -> [a]
--       append [a1, a2] [a3, a4, a5] =
--         [a1, a2, a3, a4, a5]
--       @@
--       recArg=0
--   @}
append :: [a] -> [a] -> [a]
append l1 l2 = _

> (ok) case l1 of
>       [] -> l2
>       (:) a1 l3 -> (:) a1 (append l3 l2)
```

Figure 9. Example Scythe runs for `stutter` and `append`

much more appealing to use. For example, Figure 9 shows how the system only requires a single example to synthesize the `stutter` and `append` functions over lists. Note how the examples (1) in contrast to `fromMaybe`, don't so obviously imply the implementation of the function and (2) resemble the examples that you might naturally write in documentation.

The Power of Free Theorems The guarantees of polymorphic types are well-known [26] and are one of the primary motivations for our work in automating type-directed programming with synthesis techniques. Parametricity tells us that the very shape of a polymorphic type determines very precisely the set of programs that inhabit that type. We witness this in Scythe when we provide *no* examples to the system and rely on raw-term enumeration.[5] For example, here is the output from Scythe when no examples are given for `fromMaybe`:

```
> (?) case m1 of
>       Nothing -> s1
>       Just a1 -> a1
> (?) case m1 of
>       Nothing -> s1
>       Just a1 -> s1
> (?) s1
```

The question marks in the output denote that while each reported program has the right type, they have not been found to be consistent with any examples (since we have not provided any!). However, there are so few programs to report due to polymorphism, the user can simply review each implementation to see if it matches the behavior they want for the function. This stands in contrast to the shallower search that GHC's hole-filling mechanism conducts which only suggests `s1` as a valid completion.[6]

It might be the case that some polymorphic function of interest is too complicated to be specified with examples. This example shows that synthesis techniques like those used by Scythe still have the potential to provide meaningful feedback, even when only types drive the synthesis process.

5 Further Extensions

A constraint-based approach to program synthesis opens the doors to a number of additional features that can greatly expand the scope and power of the system. We briefly reflect on them, suggesting next steps forward based on the techniques developed in this paper.

Existential Types One glaring omission from the system described in Figure 4 is the lack of existential types. In Haskell, existential types are encoded as universal types in constructors that are not mentioned in the output type of the constructor. The classic example of this are a type used to box elements of a heterogeneous list:

```
data Obj where Obj :: Show a => a -> Obj
```

From this, we can create our heterogeneous list from a regular list of `Obj`, *e.g.*, `[Obj 1, Obj "hi", Obj True]`. With this set up, we have no way of extracting the values from their `Obj` wrappers—the `Obj` effectively seals the type variable `a` from the outside world—but the typeclass constraint allows us to traverse and `show` the values.

When working with GADT types as in refine-gadt-case and refine-gadt-data, we must take care to identify which of the type variables are truly universal (that appear in the output type of the constructor) versus those that are existential (that do not appear in the output type of the constructor). For example, if we were pattern matching on a value with type `Obj`, its sole match would contain a binder for the existentially bound variable `a`. However, we must ensure that we do not unify that variable with a concrete type as its type is really sealed! This can be accomplished by calculating via a free variable check which type variables are universal or existential and then skolemizing the existential variables so that they cannot participate in the unification process.

On top of this, we must also decide how to deal with examples at existential type. Can we get away with representing

[5] By default Scythe limits the depth of nested function application and cases to avoid exponential blow-up in the search space.

[6] As of GHC version 8.6.4.

existential types with polymorphic constants and relying on skolemization to assign those constants appropriate types during synthesis or do we need a new example form for existential values?

Typeclass Constraints GADTs are only one way that type constraints are introduced in a Haskell program. More common are *typeclass constraints*, for example, the standard association list `lookup` function of type

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

constrains its first argument of type `a` to be an instance of the `eq` typeclass which in turns provides the method `(==)` to compare `a`s with. Typeclasses are ubiquitous in Haskell code, especially in highly polymorphic code that mixes typeclass constraints with multiparamteter typeclasses and functional dependencies [10].

While seemingly unrelated at first glance, Vytiniotis *et al.* demonstrate how the problems of typeclass constraints and GADTs inference can be solved through their constraint-based type inference system, OutsideIn(X) and implication constraints [25]. In order to add typeclass constraints to our synthesis system, we will likely need to adapt more of their constraint representation and solving techniques.

Partial Type Information One of the goals of this paper is taking the first steps towards understanding how to marry together constraint-based type inference techniques with program synthesis. However, oddly enough, even though we've adopted a constraint-based approach to synthesis in this work, we still have made the problems of type inference and program synthesis rather separate! In particular, because we require that the top-level binding of our synthesis problem possesses a fully annotated type, we only need to perform type inference in a very limited setting—instantiating polymorphic types during term generation—which greatly simplified our approach.

A more ambitious marriage of constraint-based type inference and program synthesis would not only leverage constraint-based reasoning techniques in synthesis but truly intertwine type inference and program synthesis. This would involve removing restrictions of having concrete types everywhere in the system (*e.g.*, the scrutinee of refine-case) and reasoning more about how to incrementally refine goal types in tandem with program discovery. Such a change would likely impact performance of the synthesizer significantly but give greater flexibility to the user in terms of specifying a program of interest when the types become complicated.

6 Related Work

Program Synthesis with Types Program synthesis, the automatic generation of programs from specification, is an enormous field encompassing the programming languages, formal verification, software engineering, and most recently the machine learning communities [8]. Many approaches are used to tackle this problem in a variety of contexts from general-purpose programming to highly-specific domains. Utilizing types as a guiding principle for synthesis is a small, yet important, piece of the puzzle, especially as it pertains to richly-typed functional programming languages and type-directed programming. These approaches span the simply-typed, general-purpose domain [1, 19], components [4, 5, 9, 12], and more richly-typed domains [6, 22]. Many of these prior efforts handle polymorphism but in an ad hoc manner or using an "enumerate all possible types" approach. The approach that we present in the paper is the first, to our knowledge, to attempt an "infer while enumerate" approach to type instantiation.

Type Inference and GADTs Our adaption of type inference is heavily rooted in the constraint-based approached proposed by Odersky *et al.* with their $HM(X)$ framework [14] as well as the presentation of constraint-based typing found in *TAPL* [20].

GADT type inference, while an undecidable problem [24], has enjoyed continuous refinement in order to develop inference algorithms that are tractable but also produce useful results [3, 10, 11, 13, 23–25]. While our system does not perform GADT type inference directly, its approach to GADT type checking and constraint passing is inspired by Vytiniotis's OutsideIn(X) framework [25].

Live Programming Our exploration of constraint-based, type-directed program synthesis lives in the context of live programming support for type-directed programming. This exploration has begun to take shape in the PL community in several forms. First is the support for language servers which provide efficient compilation services to live programming tools [2]. The second is support for hole-based development that was exclusively found in dependently-typed programming languages such as Coq, Agda, and Idris, but are now found in languages like Haskell [7]. The third is theoretical explorations of hole-based, interactive development [15, 16].

Acknowledgments

We thank the TyDe 2019 reviewers for their valuable feedback in improving this manuscript and the students of the Grinnell Pioneer PL research group that have worked on Scythe in its various forms over the last few years: Bogdan Abaev, List Berkowitz, Kevin Connors, Shelby Frazier, Reily Grant, Andrew Mack, Griffin Mareske, Ankit Pandey, Dhruv Phumbhra, Jonathan Sadun, Zachary Segall, and Zachary Susag.

This material is based upon work supported by the National Science Foundation under Grant No. 1651817. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Lennart Augustsson. 2004. [Haskell] Announcing Djinn, Version 2004-12-11, a Coding Wizard.

[2] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: A Language Server for OCaml (Experience Report). *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 103:1–103:15. https://doi.org/10.1145/3236798

[3] Sheng Chen and Martin Erwig. 2016. Principal Type Inference for GADTs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM Press, St. Petersburg, FL, USA, 416–428. https://doi.org/10.1145/2837614.2837665

[4] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 599–612. https://doi.org/10.1145/3009837.3009851

[5] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM Press, Portland, OR, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[6] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, St. Petersburg, FL, USA. https://doi.org/10.1145/2914770.2837629

[7] Matthías Páll Gissurarson. 2018. Suggesting Valid Hole Fits for Typed-Holes (Experience Report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM Press, St. Louis, MO, USA, 179–185. https://doi.org/10.1145/3242744.3242760

[8] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[9] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, Vol. 48. ACM Press, New York, NY, USA, 27–38. https://doi.org/10.1145/2499370.2462192

[10] Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg, 97–136.

[11] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. GADTs Meet Their Match: Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM Press, Vancouver, BC, Canada, 424–436. https://doi.org/10.1145/2784731.2784748

[12] Susumu Katayama. 2012. An Analytical Inductive Functional Programming System That Avoids Unintended Programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM 2012)*. ACM Press, Philadelphia, Pennsylvania, USA, 43. https://doi.org/10.1145/2103746.2103758

[13] Gabriela Moreira, Cristiano Vasconcellos, and Rodrigo Ribeiro. 2018. Type Inference for GADTs, Outsidein and Anti-Unification. In *Proceedings of the XXII Brazilian Symposium on Programming Languages (SBLP 2018)*. ACM Press, Sao Carlos, Brazil, 51–58. https://doi.org/10.1145/3264637.3264644

[14] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory and Practice of Object Systems* 5, 1 (Jan. 1999), 35–55. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4

[15] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–32. https://doi.org/10.1145/3290327

[16] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 86–99. https://doi.org/10.1145/3009837.3009900

[17] Peter-Michael Osera. 2015. *Program Synthesis with Types*. PhD Thesis. University of Pennsylvania, Philadelphia, PA.

[18] Peter-Michael Osera. 2019. Constraint-Based Type-Directed Program Synthesis. *arXiv:1907.03105 [cs]* (July 2019). arXiv:cs/1907.03105

[19] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM Press, Portland, OR, USA, 619–630. https://doi.org/10.1145/2737924.2738007

[20] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

[21] Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

[22] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. ACM Press, Santa Barbara, CA, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[23] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and Decidable Type Inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*. ACM Press, Edinburgh, Scotland, 341. https://doi.org/10.1145/1596550.1596599

[24] Vincent Simonet and François Pottier. 2007. A Constraint-Based Approach to Guarded Algebraic Data Types. *ACM Transactions on Programming Languages and Systems* 29, 1 (Jan. 2007), 1–es. https://doi.org/10.1145/1180475.1180476

[25] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

[26] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA 1989)*. ACM Press, Imperial College, London, United Kingdom, 347–359. https://doi.org/10.1145/99370.99404