Functional Program Synthesis from Linear Types

Rodrigo Mesquita and Bernardo Toninho

Nova School of Science and Technology

Abstract. Type-driven program synthesis is concerned with automatic generation of programs that satisfy a given specification, formulated as a type. One of the key challenges of program synthesis lies in finding candidate solutions that adhere to both the specification and the user's intent in an actionable amount of time. In this work, we explore how linear types allow for precise specifications suitable for synthesis, and present a framework for synthesis with linear types that, through the Curry-Howard correspondence, leverages existing proof-search techniques for Linear Logic to efficiently find type-correct programs.
We implement the synthesis framework both as a standalone language which supports classical linear types extended with recursive algebraic data types, parametric polymorphism and basic refinements; and as a GHC type-hole plugin that synthesises expressions for Haskell program holes, using the recently introduced linear types feature – showing it can generate precise solutions, remarkably fast.

Keywords: program synthesis, linear logic, propositions-as-types, proof theory

1 Introduction

Program synthesis is the automated or semi-automated process of deriving a program, i.e. generating code, from some (high-level) specification. Synthesis can be seen as a means to improve programmer productivity and program correctness (e.g. through suggestion and autocompletion). Specifications can take many forms such as natural language [7], examples [8] or rich types such as polymorphic refinement types [18] or graded types [11]. Regardless of the specification form, program synthesis must address two main sources of complexity – searching over the space of valid programs, and interpreting user intent.

Type-driven synthesis leverages rich types to make specifications more expressive and prune the valid programs search space, while maintaining a "familiar" specification interface (types) for the user. For instance, the type $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$ can be viewed as a specification, but there are an infinite number of functions that instance this type/satisfy this specification – it is extremely imprecise. On the other hand, the refinement type $(x{:}\mathsf{Int}) \to (y{:}\mathsf{Int}) \to \{z{:}\mathsf{Int} \mid z = x + y\}$ precisely specifies a function that adds its two arguments.

Linear types are another form of rich types that constrains resource usage in programs by *statically* limiting the number of times certain resources can be used during their lifetime; in particular, linear resources must be used *exactly once*. Linearity allows us to concisely describe interesting functions, since we can easily specify which arguments must be used exactly once in the body of a function. For example, the type of the linear map function (using Haskell syntax), $map :: (a \multimap b) \rightarrow [a] \multimap [b]$, specifies a function that, given a *linear* function from a to b, must consume the list of as exactly once to produce a list of bs, which can only be done by applying the function to each element. A linearity-aware program synthesizer can take the map type as a specification to unambiguously produce:

```
map f ls = case ls of
  [] → []
  (x : xs) → f x : map f xs
```

Another example is the more challenging $array :: Int \rightarrow [(Int, a)] \rightarrow Array\ a$ goal taken from Linear Haskell [2], which is implemented in terms of a linear interface to mutable arrays. Remarkably, that linear interface is also precise enough that our framework is capable of synthesizing the correct implementation (§ 4).

However, it is not at all obvious how to automate such a synthesis procedure in a general setting where functions can make use of recursion, algebraic data types and pattern matching. For instance, any naive approach that simply iterates over all possible programs (of which there are infinitely many) and checks them against the given specification (i.e., type-checking) would be very unlikely to find a function that matches the user intent in a reasonable time frame.

Synthesis with linear types, combined with other advanced typing features, has generally been overlooked in the literature, despite their long-known potential [20,4,2] and strong proof-theoretic foundations [1,6,5]. One aspect that makes linear types particularly appealing from the point of view of program synthesis is how linearity can affect the search space of valid programs: all programs where a linear resource is used non-linearly (i.e. not exactly once) are ill-typed and can be discarded. With linearity built into the synthesis process, usage of a linear variable more than once is not considered, and unused variables are identified during synthesis, constraining the space of valid programs.

In this work we explore type-based synthesis of functional programs using linear types under the lens of the Curry-Howard correspondence. Notably, we employ techniques from linear logic *proof search* as a mechanism for program synthesis, leveraging the proofs-as-programs connection between linearly typed functional programs and linear logic proofs. Our contributions are as follows:

- We present a framework for synthesis of functional programs (§ 2.1) from specifications based on linear types, leveraging established proof-search techniques for linear logic under the lens of the Curry-Howard isomorphism. Specifically, the core of the synthesis procedure is a *sound* and *complete* system consisting of *bottom-up* proof-search in propositional linear logic, using a technique called *focusing* [1]. Our approach, being grounded in propositions-as-types, ensures that all synthesized programs (i.e. proofs) are well-typed *by construction* (i.e. if the synthesis procedure produces a program, then the program intrinsically satisfies its specification).

- We extend the core synthesis framework and language with algebraic data types, recursive functions, parametric polymorphism and type refinements. These extra-logical extensions require us to abandon completeness and to develop techniques to effectively explore the search space in the presence of recursion (§ 3).
- We present two implementations of our synthesis framework [**?**,**?**], one as a GHC plugin that synthesizes expressions for Linear Haskell [2] program holes, the other in a standalone language with the same features the synthesis process supports, and benchmark them on diverse synthesis goals (§ 4).

2 Synthesis as Proof Search

The Curry-Howard correspondence [21] describes the fundamental connection between logic and programming languages: propositions are types, and proofs are programs. Under this lens, we can view *bottom-up* proof-search as *program synthesis* – starting from a goal proposition A, finding a proof of A *is exactly* the process of generating a program of type A.

Typically, the Curry-Howard correspondence is developed between so-called systems of natural deduction and core functional languages such as the λ-calculus, where logical rules have a direct, one-to-one, mapping to typing rules. However, even though proofs in natural deduction can be interpreted as programs, a natural deduction proof system does not directly describe an algorithm for proof search. An example that highlights this is the *modus ponens* rule from intuitionistic logic (below on the left) and its analogous *function application* typing rule from the simply-typed λ-calculus (below on the right):

$$\frac{\Gamma \vdash \alpha \rightarrow \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta}\ (\textsc{mp}) \qquad\qquad \frac{\Gamma \vdash M : \alpha \rightarrow \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash M\ N : \beta}\ (\rightarrow \mathrm{E})$$

If we interpret the rules *bottom-up* we can see the modus ponens rule as "to find a proof of β under assumptions Γ, find a proof of $\alpha \rightarrow \beta$ and a proof of α with the same assumptions, for some α". However, an algorithm based on these rules would have to invent an α for which the proof can be completed, with no obvious relation between α and the goal β. In essence, inference rules in natural deduction are ill-suited for bottom-up proof-search since not all rules have an algorithmic bottom-up reading. A more suitable candidate for bottom-up proof search system is the equivalent *sequent calculus* in which all inference rules can be naturally read in a bottom-up manner. The corresponding implication elimination (or *function application*) rule is instead:

$$\frac{\Gamma, f{:}\alpha \rightarrow \beta, x{:}\beta \vdash M : \tau \qquad \Gamma, f{:}\alpha \rightarrow \beta \vdash N : \alpha}{\Gamma, f{:}\alpha \rightarrow \beta \vdash M[f\ N/x] : \tau}\ (\rightarrow L)$$

which can be understood *bottom-up*, through the lens of synthesis this time, as "to synthesize an expression of type τ when $f{:}\alpha \rightarrow \beta$ is in the context, synthesize an argument N of type α, and an expression $M{:}\tau$ assuming $x{:}\beta$ in the context – then replace occurrences of x by $f\ N$ in M". Nevertheless, a sequent calculus is still not completely suited for proof search due to non-determinism in selecting which rules to apply (e.g. if multiple function types are available in the context, which should we attempt to use?).

Andreoli's focusing [1] is a technique that further disciplines (linear logic) proof-search by reformulating the rules of the sequent calculus. Focusing reduces the non-determinism inherent to proof search by leveraging the fact that the *order* in which certain rules are applied does not affect the outcome of the search, and by identifying the non-determinism in the search process that pertains to true unknowns (i.e., rules whose application modifies what can be subsequently proved) – marking precisely the branches of the search space that may need to be revisited (i.e., backtracked to). A *focused sequent calculus* can hence be effectively read as a procedure for proof search for (a significant fragment of) linear logic that turns out to be both *sound* and *complete* – a sequent is provable if and only if it is derivable in the focused system.

Our core synthesis framework thus comprises of a reading of focused proof search in (intuitionistic) linear logic, where proofs are seen as programs in a linearly-typed λ calculus. In linear logic, propositions are interpreted as resources that are *consumed* during the inference process. Where in standard propositional logic we are able to use an assumption as many times as we want, in linear logic every resource (i.e., every assumption) must be used *exactly once*, or *linearly*. In the remainder of this work we will move interchangeably from linear propositions to linear *types*. As an example, consider the typing rules for the linear function ($\multimap$), the linear counterpart to the standard function type; linear product ($\otimes$), related to the standard product type; and linear variables:

$$\frac{\Delta, x{:}A \vdash M : B}{\Delta \vdash \lambda x.M : A \multimap B}\ (\multimap \mathrm{R}) \qquad \frac{\Delta_1 \vdash M : A \qquad \Delta_2 \vdash N : B}{\Delta_1, \Delta_2 \vdash (M, N) : A \otimes B}\ (\otimes\ \mathrm{R}) \qquad \frac{}{x : A \vdash x : A}\ (\textsc{var})$$

The rules define the judgment $\Delta \vdash M : A$, stating that term M has type A using linear variables in Δ. Rule ($\multimap$ R) explicates that to type a λ-abstraction $\lambda x.M$ with type $A \multimap B$, the body M must use x exactly once with type A to produce B. Note how the variable rule enforces the exact usage since no other ambient variables are allowed. This is also observed in the ($\otimes$ R) rule, which states that to type the linear pair (M, N) with $A \otimes B$, the available resources must be split in two regions (Δ_1 and Δ_2), one that is used in M and the other, disjointly, in N (the logical rules can be obtained by omitting the terms). Unlike in standard propositional logic, where assumptions can be weakened and contracted (i.e., discarded and duplicated) and so can simply be maintained globally as they are introduced in a derivation, in linear logic assumptions cannot be weakened or contracted and thus a system of resource management [5,12] is combined with focusing in order to algorithmically track linear resource usage.

We now present in detail the techniques that make up our synthesis framework based on linear types. We first introduce the core of the synthesizer (§ 2.1) which is a more or less direct interpretation of focusing as proof search; which we soundly extend (§ 3) with extra-logical but programming-centric features such as general recursion and abstract data types (necessarily abandoning completeness). We note that a *sound* set of rules guarantees cannot synthesize ill-typed programs; and that the valid programs derivable through them reflect the subjective trade-offs we committed to in order to produce an effective synthesizer.

2.1 Core Language

Core Rules. The core language of our framework is a simply-typed linear λ-calculus with linear functions ($\multimap$), additive ($\&$) and multiplicative ($\otimes$) pairs (denoting alternative and simultaneous occurrence of resources, respectively), multiplicative unit ($\mathbf{1}$), additive sums ($\oplus$) and the exponential modality ($!$), which internalizes unrestricted use of variables. The syntax of terms (M, N) and types (τ, σ) is given below (we highlight the type for which the terms in a given line form the introduction and elimination forms, respectively):

$$
\begin{array}{lcll}
M, N & ::= & u, v & \\
& \mid & \lambda x.M \mid M\,N & (\multimap) \\
& \mid & M \,\&\, N \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M & (\&) \\
& \mid & M \otimes N \mid \mathsf{let}\ u \otimes v = M\ \mathsf{in}\ N & (\otimes) \\
& \mid & \star \mid \mathsf{let}\ \star = M\ \mathsf{in}\ N & (\mathbf{1}) \\
& \mid & \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid (\mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ u \Rightarrow N_1 \mid \mathsf{inr}\ v \Rightarrow N_2) & (\oplus) \\
& \mid & !M \mid \mathsf{let}\ !u = M\ \mathsf{in}\ N & (!) \\
\tau, \sigma & ::= & a \mid \tau \multimap \sigma \mid \tau \,\&\, \sigma \mid \tau \otimes \sigma \mid \mathbf{1} \mid \tau \oplus \sigma \mid\ !\tau &
\end{array}
$$

In intuitionistic sequent calculi, each connective has a so-called *left* and a *right* rule, which effectively define how to decompose an ambient assumption of a given proposition and how to prove a certain proposition is true, respectively. In a *focused* sequent calculus we further identify so-called invertible and non-invertible inference rules. Andreoli [1] observed that the connectives of linear logic can be divided into two categories, dubbed synchronous and asynchronous. Asynchronous connectives are those whose right rules are *invertible*, i.e. they can be applied eagerly during proof search without altering provability (and so the order in which these rules are applied is irrelevant) and whose left rules are not invertible. Synchronous connectives are dual. The asynchronous connectives are $\multimap$ and $\&$ and the synchronous ones are $\otimes, \mathbf{1}, \oplus, !$.

Given this separation, focusing divides proof search into two alternating phases: the inversion phase, in which we apply *all* invertible rules eagerly, and the focusing phase, in which we decide a proposition to *focus* on, and then apply non-invertible rules, staying *in focus* until we reach an asynchronous/invertible proposition, the proof is complete, or no rules are applicable, in which case the proof must *backtrack* to the state at which the focusing phase began. As such, with focusing, the linear sequent calculus judgment $\Gamma; \Delta \vdash M : A$, meaning that $M : A$ is derivable from the linear assumptions in Δ and non-linear assumptions in Γ, is split into four judgments, grouped into the two phases.

For the invertible phase, a context Ω holds propositions that result from decomposing connectives. The right inversion and left inversion judgments are written $\Gamma; \Delta; \Omega \vdash M : A \Uparrow$ and $\Gamma; \Delta; \Omega \Uparrow \vdash M : A$, where the $\Uparrow$ indicates the connective or context being inverted. For the focusing phase (all non-invertible rules can apply), the proposition under focus can be the goal or one in Γ or Δ. The right focus judgment is written $\Gamma; \Delta \vdash M : A \Downarrow$ and the left focus judgment $\Gamma; \Delta; B \Downarrow \vdash M : A$, where $\Downarrow$ indicates the proposition under focus.

As alluded in the previous section, to handle the context splitting required to prove subgoals, we augment the judgments above using Hodas and Miller's resource management technique [5,12] where a pair of input/output linear contexts is used to propagate the yet unused linear resources across subgoals; e.g. the left inversion judgment is written $\Gamma; \Delta/\Delta'; \Omega \Uparrow \vdash M : A$ where Δ is the input linear context and Δ' is the output one.

Combining linear logic (i.e., the linear lambda calculus through the Curry-Howard correspondence), resource management, and focusing, we obtain the following core formal system[1] (inspired by [6,17]) – in which the rule $\multimap$R is read: to synthesize a program of type $A \multimap B$ while inverting right (the $\Uparrow$ on the goal), with unrestricted context Γ, linear context Δ, and inversion context Ω, assume $x{:}A$ in Ω to synthesize a program M of type B, and return $\lambda x.M$. We begin with right invertible rules, which decompose the goal until it becomes a synchronous proposition:

$$\frac{\Gamma; \Delta/\Delta'; \Omega, x{:}A \vdash M : B \Uparrow \qquad x \notin \Delta'}{\Gamma; \Delta/\Delta'; \Omega \vdash \lambda x.M : A \multimap B \Uparrow}\ (\multimap R)$$

When we reach a non-invertible proposition on the right, we start inverting the Ω context. The rule to transition to inversion on the left is:

$$\frac{\Gamma; \Delta/\Delta'; \Omega \Uparrow \vdash M : C \qquad C \text{ not right asynchronous}}{\Gamma; \Delta/\Delta'; \Omega \vdash M : C \Uparrow}\ (\Uparrow\mathrm{R})$$

We then apply left invertible rules for asynchronous connectives, which decompose asynchronous propositions in Ω:

$$\frac{\Gamma; \Delta/\Delta'; \Omega, y{:}A, z{:}B \Uparrow \vdash M : C \qquad y, z \notin \Delta'}{\Gamma; \Delta/\Delta'; \Omega, x{:}A \otimes B \Uparrow \vdash \text{ let } y \otimes z = x \text{ in } M : C}\ (\otimes L)$$

$$\frac{\Gamma, y{:}A; \Delta/\Delta'; \Omega \Uparrow \vdash M : C}{\Gamma; \Delta/\Delta'; \Omega, x{:}!A \Uparrow \vdash \text{ let } !y = x \text{ in } M : C}\ (!L)$$

When we find a synchronous (i.e. non-invertible) proposition in Ω, we simply move it to the linear context Δ, and keep inverting on the left:

$$\frac{\Gamma; \Delta, x : A/\Delta'; \Omega \Uparrow \vdash M : C \qquad A \text{ not left asynchronous}}{\Gamma; \Delta/\Delta'; \Omega, x : A \Uparrow \vdash M : C}\ (\Uparrow\mathrm{L})$$

[1] For the sake of brevity, we've omitted some rules such as those for the additive pair and disjunction.

After inverting all the asynchronous propositions in Ω we reach a state where there are no more propositions to invert ($\Gamma'; \Delta'; \cdot \Uparrow \vdash C$). At this point, we want to *focus* on a proposition. The focus object will be: the proposition on the right (the goal), a proposition from the linear Δ context, or a proposition from the unrestricted Γ context. For these options we have three *decision* rules:

$$\frac{\Gamma; \Delta/\Delta' \vdash M : C \Downarrow \qquad C \text{ not atomic}}{\Gamma; \Delta/\Delta'; \cdot \Uparrow \vdash M : C} \ (\textsc{decide}\text{R})$$

$$\frac{\Gamma; \Delta/\Delta'; x : A \Downarrow \vdash M : C}{\Gamma; \Delta, x : A/\Delta'; \cdot \Uparrow \vdash M : C} \ (\textsc{decide}\text{L}) \qquad \frac{\Gamma, A; \Delta/\Delta'; A \Downarrow \vdash M : C}{\Gamma, A; \Delta/\Delta'; \cdot \Uparrow \vdash M : C} \ (\textsc{decide}\text{L!})$$

The decision rules are followed by either left or right focus rules. To illustrate, consider the $\multimap$L left focus rule. The rule states that to produce a program of type C while left focused on the function x of type $A \multimap B$, we first check that we can produce a program of type C by using B. If this succeeds in producing some program M, this means that we can apply x to solve our goal. We now synthesize a program N of type A, switching to the right inversion judgment ($\Uparrow$). To construct the overall program, we replace in M all occurrences of variable y with the application $x\,N$. The remaining left rules follow a similar pattern. The right focus rules are read similarly to right inversion ones, albeit the goal and sub-goals are under focus (except for $!R$).

$$\frac{\Gamma; \Delta/\Delta'; y{:}B \Downarrow \vdash M : C \qquad \Gamma; \Delta'/\Delta''; \cdot \vdash N : A \Uparrow}{\Gamma; \Delta/\Delta''; x{:}A \multimap B \Downarrow \vdash M\{(x\,N)/y\} : C} \ (\multimap L)$$

$$\frac{\Gamma; \Delta/\Delta' \vdash M : A \Downarrow \qquad \Gamma; \Delta'/\Delta'' \vdash N : B \Downarrow}{\Gamma; \Delta/\Delta'' \vdash (M \otimes N) : A \otimes B \Downarrow} \ (\otimes R) \qquad \frac{}{\Gamma; \Delta/\Delta \vdash \star : \mathbf{1} \Downarrow} \ (\mathbf{1}R)$$

Eventually, the focus proposition will no longer be synchronous, i.e. it's atomic or asynchronous. If we're left focused on an atomic proposition we either instantiate the goal or fail. Otherwise the left focus is asynchronous and we can start inverting it. If we're right focused on a proposition that isn't right synchronous, we switch to inversion as well. Three rules model these conditions:

$$\frac{}{\Gamma; \Delta/\Delta'; x{:}A \Downarrow \vdash x : A} \ (\textsc{init}) \qquad \frac{\Gamma; \Delta/\Delta'; \cdot \vdash M : A \Uparrow}{\Gamma; \Delta/\Delta' \vdash M : A \Downarrow} \ (\Downarrow R)$$

$$\frac{\Gamma; \Delta/\Delta'; x : A \Uparrow \vdash M : C \qquad A \text{ not atomic and not left synchronous}}{\Gamma; \Delta/\Delta'; x : A \Downarrow \vdash M : C} \ (\Downarrow L)$$

The rules presented above make the core of our synthesizer. As a proof system, focusing is both sound and complete – a sequent is provable in the focused system if and only if it is provable in linear logic. We note however, that proof search in full linear logic is *undecidable* [13].

To illustrate the core synthesis framework, consider the goal $A \otimes B \multimap B \otimes A$. Starting focused on the goal, we can construct a derivation (i.e. a program) by identifying the rules that are applicable at any given moment. If more than one rule is applicable, we must make a non-determinisic choice, but focusing guarantees those choices are only required for "true unknowns". A derivation for this goal can be constructed by applying, from the bottom-up, $\multimap R, \Uparrow R, \otimes L, \Uparrow L, \Uparrow L$, DECIDER, $\otimes R$, and then instantiating both the sub-goals B and A using $\Downarrow R, \Uparrow R, \Uparrow L$, DECIDEL, INIT. Note that many of these rules don't intrinsically change the proof, but are necessary in the proof-search procedure to eliminate non-essential non-determinism. The program corresponding to the proof is $\lambda x.\ \mathsf{let}\ (a, b) = x\ \mathsf{in}\ (b, a)$. We leave writing the derivation as an exercise.

3 Beyond Propositional Logic

By itself, the core synthesis process can only output simple non-recursive programs. In this section, we extend our framework to be able to synthesize more interesting programs featuring general recursion over algebraic data types (ADTs) and polymorphism. The combination of these features diverges from the pure logical interpretation of focusing since unguarded general recursion is unsound from a logical perspective (and decomposing ADTs through pattern matching is uncommon in proof theory).

In its simplest form, an algebraic data type (ADT) is a tagged sum of any type (i.e. a named type that can be instantiated by one of many tags, or constructors) that take some value of a fixed type. In general, since the tagged types can be products ($A \otimes B$), or the unit ($\mathbf{1}$), constructors can have an arbitrary fixed number of parameters. The grammar of our core calculus is extended as (where C_n is a constructor for values of some type T):

$$M, N ::= \ldots \mid C_n\ M \mid (\mathsf{case}\ M\ \mathsf{of}\ \ldots \mid C_n\ u \Rightarrow N) \qquad \tau, \sigma ::= \ldots \mid T$$

Algebraic data types are related to the $(\oplus)$ type – both are forms of disjunction. There is a right rule for each constructor of the data type, requiring only that a term is synthesized for the argument of the constructor; and there is one left rule to deconstruct a value of ADT type in the context by pattern matching, requiring a term of the same type to be synthesized for each possible branch. Naively, one might consider the rules:

$$\frac{\Gamma; \Delta/\Delta' \vdash M : X_n \Downarrow}{\Gamma; \Delta/\Delta' \vdash\ C_n\ M : T \Downarrow}\ (\textsc{adt}R)$$

$$\frac{\overline{\Gamma; \Delta/\Delta'_n; \Omega, y_n{:}X_n \Uparrow \vdash M_n : C} \qquad \overline{y_n \notin \Delta'_n} \qquad \Delta'_1 = \Delta'_2 = \cdots = \Delta'_n}{\Gamma; \Delta/\Delta'_1; \Omega, x{:}T \Uparrow \vdash\ \mathsf{case}\ x\ \mathsf{of}\ \ldots \mid C_n\ y_n \to M_n : C}\ (\textsc{adt}L)$$

However, for recursively defined data types, i.e. for constructors that take as an argument a value of the type they construct, a direct application of the rules above will not terminate. Consider, for example, type T and its sole constructor C_1. When synthesizing a derivation for a goal $T \multimap D$, for some D, we could infinitely apply ADTL:

$$\frac{\dfrac{\dfrac{\cdots}{\Gamma; \Delta/\Delta'; \Omega, y{:}T \Uparrow \vdash \mathsf{case}\ y\ \mathsf{of}\ C_1\ z \to \cdots : D}\ (\textsc{adt}L)}{\Gamma; \Delta/\Delta'; \Omega, x{:}T \Uparrow \vdash \mathsf{case}\ x\ \mathsf{of}\ C_1\ y \to \cdots : D}\ (\textsc{adt}L)}{\Gamma; \Delta/\Delta'; \Omega, x{:}T \vdash \cdots : D \Uparrow}\ (\Uparrow R)$$

Symmetrically, the derivation for goal T is also infinite, since we can apply $\textsc{adt}$R infinitely, never closing the proof:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma; \Delta/\Delta'; \Omega \vdash C_1 \cdots : T \Downarrow}\,(\textsc{adt}\mathrm{R})}{\Gamma; \Delta/\Delta'; \Omega \vdash C_1 \cdots : T \Downarrow}\,(\textsc{adt}\mathrm{R})}{\Gamma; \Delta/\Delta'; \Omega \vdash C_1 \cdots : T \Downarrow}\,(\textsc{adt}\mathrm{R})$$

To account for recursively defined types, we restrict their decomposition when synthesizing branches of a case construct, and, symmetrically, disallow construction of data types when trying to synthesize an argument for their constructors. To model this, we use two more contexts, P_C for constraints on construction and P_D for constraints on deconstruction. Together, they hold a list of data types that cannot be constructed or deconstructed at a given point, respectively. For convenience, they are represented by a single P if unused and all non-ADT rules trivially propagate these. The ADT rules account for recursion as follows, where $P'_C = P_C, T$ if T is recursive and $P'_C = P_C$ otherwise (P'_D is dual):

$$\dfrac{(P'_C; P_D); \Gamma; \Delta/\Delta' \vdash M : X_n \Downarrow \qquad T \notin P_C}{(P_C; P_D); \Gamma; \Delta/\Delta' \vdash \ C_n\ M : T \Downarrow}\,(\textsc{adt}\mathrm{R})$$

$$(\textsc{adt}\mathrm{L})\ \dfrac{\overline{(P_C; P'_D); \Gamma; \Delta/\Delta'_n; \Omega, y_n{:}X_n \Uparrow \vdash M_n : C} \qquad \overline{y_n \notin \Delta'_n} \qquad T \notin P_D \qquad \Delta'_1 = \cdots = \Delta'_n}{(P_C; P_D); \Gamma; \Delta/\Delta'_1; \Omega, x{:}T \Uparrow \vdash \ \mathsf{case}\ x\ \mathsf{of}\ \ldots\ \mid\ C_n\ y_n \to M_n : C}$$

These modifications block the infinite derivations described above. However, they also greatly limit the space of derivable programs, leaving the synthesizer effectively unable to synthesize from specifications with recursive types. To prevent this, we add two rules to complement the restrictions on construction and destruction of recursive types. First, since we can't deconstruct some ADTs any further due to these constraints, but must utilize all propositions linearly in some way, all propositions in Ω whose deconstruction is restricted are to be moved to the linear context Δ. Second, without any additional rules, an ADT in the linear context will loop back to the inversion context, jumping back and forth between the two contexts; instead, when focusing on an ADT, we should either instantiate the goal (provided they're the same type), or switch to inversion if and only if its decomposition is *not* restricted:

$$\dfrac{(P_C; P_D); \Gamma; \Delta, x{:}T/\Delta'; \Omega \Uparrow \vdash M : C \qquad T \in P_D}{(P_C; P_D); \Gamma; \Delta/\Delta'; \Omega, x{:}T \Uparrow \vdash M : C}\,(\textsc{adt}{\Uparrow}\mathrm{L})$$

$$\dfrac{(P_C; P_D); \Gamma; \Delta/\Delta'; x{:}T \Uparrow \vdash M : T \qquad T \notin P_D}{(P_C; P_D); \Gamma; \Delta/\Delta'; x{:}T \Downarrow \vdash M : T}\,(\textsc{adt}{\Downarrow}\mathrm{L})$$

Altogether, the rules above ensure that a recursive ADT will be deconstructed once, and that subsequent equal ADTs will only be useable from the linear context – essentially forcing them to be used to instantiate another proposition, which will typically be an argument for the recursive call.

To synthesize recursive functions, we can simply label the main goal as f and extend the unrestricted context with the label f of the appropriate type. That is, to synthesize a recursive function of type $A \multimap B$ named f, the initial judgment can be written as

$$\Gamma, f{:}A \multimap B; \Delta/\Delta'; \Omega \vdash M : A \multimap B \Uparrow$$

and so all subderivations will have $(f{:}A \multimap B)$ available in Γ. However, we must restrict immediate uses of the recursive call since otherwise every goal would have a trivial proof (a non-terminating function that just calls itself), shadowing relevant solutions. Instead, our framework allows the use of recursion only *after* having deconstructed a recursive ADT, satisfying the invariant: the recursive call can only be used in *recursive branches of ADT deconstruction*, i.e. the recursive call should only take "smaller" terms as arguments. We also forbid further recursive calls when synthesizing arguments for the recursive call itself.

Polymorphism. A polymorphic type, or a type *scheme*, is of the form $\forall\overline{\alpha}.\ \tau$ where $\overline{\alpha}$ is a set of variables that stand for (non-polymorphic) types in τ.

Synthesis for a scheme comprises of effectively removing the quantification, and then treating its type variables uniformly. First, type variables are considered *atomic types*, then, we instantiate the bound variables of the scheme as described by the Hindley-Milner [15,10] type instantiation rule (put simply, generate fresh names for each bound type variable); e.g. the scheme $\forall\alpha.\ \alpha \multimap \alpha$ could be instantiated to $\alpha_0 \multimap \alpha_0$, for some fresh α_0. We add such a rule to our system, where $\forall\overline{\alpha}.\ \tau \sqsubseteq \tau'$ indicates type τ' is an *instantiation* of type scheme $\forall\overline{\alpha}.\ \tau$:

$$\frac{P; \Gamma; \Delta/\Delta'; \Omega \vdash M : \tau' \Uparrow \qquad \forall\overline{\alpha}.\ \tau \sqsubseteq \tau'}{P; \Gamma; \Delta/\Delta'; \Omega \vdash M : \forall\overline{\alpha}.\ \tau \Uparrow}\ (\forall R)$$

As such, the construction of a derivation in which the only rule that can derive an atom is the INIT rule corresponds to the synthesis of a program where some expressions are treated agnostically, i.e. a polymorphic program.

The main challenge of polymorphism in synthesis is the usage of schemes from the unrestricted context. The context Γ now holds both (monomorphic) types and schemes. Consequently, after the rule DECIDELEFT! is applied, we are left-focused on either a type or a scheme. Since left focus on a type is already well defined, we need only specify how to focus on a scheme.

Our algorithm instantiates bound type variables of the focused scheme with fresh *existential* type variables, and the instantiated type becomes the left focus. Inspired by the Hindley-Milner system, we also generate inference constraints on the existential type variables (postponing the decision of what type it should be to be used in the proof), and collect them in a new constraints context Θ that is propagated across derivation branches (by having an input and output context (Θ/Θ')). In contrast to Hindley-Milner inference, new constraints are immediately solved against all other constraints – a branch of the search is desired to fail as soon as possible. Note that we instantiate the scheme with *existential* type variables $(?\alpha)$ rather than type variables (α) since the latter represent universal types during synthesis, and the former represent a concrete instance of a scheme, that might induce constraints on other type variables. Additionally, we require that all existential type variables are eventually assigned a concrete type. These concepts are formalized with the following rules, where $\forall\overline{\alpha}.\ \tau \sqsubseteq_E \tau'$ means type τ' is an *existential instantiation* of scheme $\forall\overline{\alpha}.\ \tau$, $\mathrm{ftv}_E(\tau')$ is the set of free *existential* type variables in type τ', $?\alpha \mapsto \tau_x$ is a mapping from *existential* type $?\alpha$ to type τ_x, and $\textsc{unify}(c, \Theta)$ indicates whether constraint c can be unified with those in Θ:

$$\frac{\Theta/\Theta'; P; \Gamma; \Delta/\Delta'; \tau' \Downarrow \vdash M : C \qquad \forall\overline{\alpha}.\ \tau \sqsubseteq_E \tau' \qquad \mathrm{ftv}_E(\tau') \cap \{?\alpha \mid (?\alpha \mapsto \tau_x) \in \Theta'\} = \emptyset}{\Theta/\Theta'; P; \Gamma; \Delta/\Delta'; \forall\overline{\alpha}.\ \tau \Downarrow \vdash M : C}\ (\forall L)$$

$$\frac{\textsc{unify}(?\alpha \mapsto C, \Theta)}{\Theta/\Theta, ?\alpha \mapsto C; P; \Gamma; \Delta/\Delta'; x{:}?\alpha \Downarrow \vdash x : C}\ (?L) \qquad \frac{\textsc{unify}(?\alpha \mapsto A, \Theta)}{\Theta/\Theta, ?\alpha \mapsto A; P; \Gamma; \Delta/\Delta'; x{:}A \Downarrow \vdash x :?\alpha}\ (\Downarrow ?L)$$

4 Evaluation

We implemented our framework both as a Haskell GHC plugin and as a standalone synthesizer that can typecheck Haskell-like programs with "goal signatures" for which valid expressions are synthesized. We've tested and benchmarked both implementations on numerous synthesis challenges with successful results. Among the more intricate examples, we can easily synthesize the *Monad* instances of *Maybe* and *State*. However, the more interesting result is a real-world example from [2]: with linear types one can provide a safe interface to manipulate mutable arrays. Linear Haskell [2] provides an implementation of *array* :: *Int* → [(*Int*, *a*)] → *Array a* which, internally, uses mutable arrays via the interface:

```
newMArray :: Int → (MArray a ⊸ Ur b) ⊸ b
write :: MArray a ⊸ (Int, a) → MArray a
read :: MArray a ⊸ Int → (MArray a, Ur b)
freeze :: MArray a ⊸ Ur (Array a)
```

The flagship result from our synthesis framework, which also illustrates the preciseness of linear types, is that we're able to synthesize the exact implementation of *array* given in Linear Haskell given the above interface and the *array* type goal, all in a hundred milliseconds:

```
array size pairs = newMArray size (λma → freeze (foldl write ma pairs))
```

The standalone implementation further supports (experimentally) refinement types and additional synth guidelines. Figure 1 lists benchmarks for a suite of examples. The Goal column describes the type of the synthesized term using typical Haskell terminology. The Keywords column denotes the use of additional synthesis guidance features that we implemented in our synthesizer: the *choose* keyword instructs the synthesizer to stop after one valid term is found, the equality clause in the list reverse function serves as an input output example that guides the search, the *depth* keyword controls the instantiation depth of quantifiers. The Components column describes the library of function (signatures) provided for the particular synthesis goal.

Group	Goal	Avg. time ± σ	Keywords	Components
Linear Logic	uncurry	$133\mu s \pm 4.9\mu s$		
	call by name	$196\mu s \pm 4.6\mu s$		
	0/1	$294\mu s \pm 5.3\mu s$		
List	map	$288\mu s \pm 7.2\mu s$		
	append	$292\mu s \pm 7.0\mu s$		
	foldl	$1.69ms \pm 5.3\mu s$	*choose 1*	
	foldr	$704\mu s \pm 10\mu s$		
	concat	$505\mu s \pm 18\mu s$		
	reverse	$17.4ms \pm 515\mu s$	*reverse [1,2] == [2,1]*	
Maybe	>>=	$194\mu s \pm 5.3\mu s$		
	maybe	$161\mu s \pm 4.8\mu s$		
State	runState	$190\mu s \pm 6.8\mu s$		
	>>=	$979\mu s \pm 23\mu s$		
	>>=	∞	*using (runState)*	
	get	$133\mu s \pm 3.8\mu s$		
	put	$146\mu s \pm 3.4\mu s$		
	modify	$219\mu s \pm 4.9\mu s$		
	evalState	$156\mu s \pm 4.0\mu s$		
Misc	either	$197\mu s \pm 5.3\mu s$		
Array	array 4	$80ms \pm 870\mu s$	*depth 3* *using (foldl),depth 3*	*freeze, foldl* *newMArray,write*
Refinements	add3	$39ms \pm 1.1ms$		+

Fig. 1. Benchmarks

5 Related Work

Type-based program synthesis is a vast field of study. Most works [11,18,16,8] follow some variation of the synthesis-as-proof-search approach. Focusing in synthesis appeared first in the literature in [14]. Each synthesis framework differ due to a variety of rich types explored and their corresponding logics and languages.

The work [18] also studies synthesis of recursive functional programs in an "advanced" context. Their specifications combine two rich forms of types: polymorphic and refinement types. We also support refinements (and polymorphism), but they are not as integrated in the synthesis process as in [18]. Instead, our synthesizer leverages the expressiveness of linear types and techniques for proof-search in linear logic to guide its process.

The work [11] synthesizes programs using an approach similar ours. It employs so-called graded modal types, which are a refinement of pure linear types that allows for a quantitative specification of resource usage, in contrast to ours either *linear* or *unrestricted* (via the linear logic exponential) use of assumptions. Their resource management is thus more complex than ours. They also use focusing as a solution to trim down search space and to ensure that synthesis only produces well-typed programs. However, since their underlying logic is *modal* rather than purely *linear*, it lacks a clear correspondence with concurrent session-typed programs [4,3], which is a crucial avenue of future work. Moreover, their use of grading effectively requires an SMT solver to be integrated with the synthesis procedure, which can limit the effectiveness of the overall approach. Additionally, our system extends the focusing-based system with recursion, ADTs, polymorphism and refinements to synthesize more expressive programs.

Acknowledgements This work is supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT.IP.

References

1. Andreoli, J.M.: Logic Programming with Focusing Proofs in Linear Logic. Journal of Logic and Computation **2**(3), 297–347 (06 1992). `https://doi.org/10.1093/logcom/2.3.297`, `https://doi.org/10.1093/logcom/2.3.297`
2. Bernardy, J.P., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–29 (Jan 2018). `https://doi.org/10.1145/3158093`, `http://dx.doi.org/10.1145/3158093`
3. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR 2010. pp. 222–236 (2010)
4. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
5. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theor. Comput. Sci. **232**(1-2), 133–163 (2000). `https://doi.org/10.1016/S0304-3975(99)00173-5`, `https://doi.org/10.1016/S0304-3975(99)00173-5`
6. Chaudhuri, K., Pfenning, F.: A focusing inverse method theorem prover for first-order linear logic. In: Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings. pp. 69–83 (2005). `https://doi.org/10.1007/11532231\_6`, `https://doi.org/10.1007/11532231_6`
7. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code (2021)
8. Frankle, J., Osera, P., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 802–815. ACM (2016). `https://doi.org/10.1145/2837614.2837629`, `https://doi.org/10.1145/2837614.2837629`
9. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society **146**, 29–60 (1969), `http://www.jstor.org/stable/1995158`
10. Hughes, J., Orchard, D.: Resourceful program synthesis from graded linear types. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12561, pp. 151–170. Springer (2020). `https://doi.org/10.1007/978-3-030-68446-4\_8`, `https://doi.org/10.1007/978-3-030-68446-4_8`
11. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theor. Comput. Sci. **410**(46), 4747–4768 (2009). `https://doi.org/10.1016/j.tcs.2009.07.041`, `https://doi.org/10.1016/j.tcs.2009.07.041`

12. Lincoln, P., Scedrov, A., Shankar, N.: Decision problems for second-order linear logic. In: Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995. pp. 476–485. IEEE Computer Society (1995). https://doi.org/10.1109/LICS.1995.523281, https://doi.org/10.1109/LICS.1995.523281
13. Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. Proc. ACM Program. Lang. **4**(ICFP) (aug 2020). https://doi.org/10.1145/3408991, https://doi.org/10.1145/3408991
14. Mesquita, R.: The sili synthesiser - synthesis of linear functional programs. https://github.com/alt-romes/slfl (2021)
15. Mesquita, R.: A ghc plugin for synthesizing haskell programs from linear types using bottom-up proof search in linear logic with focusing. https://github.com/alt-romes/ghc-linear-synthesis-plugin (2022)
16. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://doi.org/10.1016/0022-0000(78)90014-4, https://doi.org/10.1016/0022-0000(78)90014-4
17. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 619–630. ACM (2015). https://doi.org/10.1145/2737924.2738007, https://doi.org/10.1145/2737924.2738007
18. Pfenning, F.: Focus handouts. https://www.cs.cmu.edu/~fp/courses/15816-f01/handouts/focus.pdf
19. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 522–538 (2016). https://doi.org/10.1145/2908080.2908093, https://doi.org/10.1145/2908080.2908093
20. Wadler, P.: Linear types can change the world! In: PROGRAMMING CONCEPTS AND METHODS. North (1990)
21. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). https://doi.org/10.1145/2699407, https://doi.org/10.1145/2699407