Synthesis of Linear Functional Programs

Rodrigo Mesquita - N^o 55902
Supervisor: Bernardo Toninho

NOVA School of Science and Technology

1 Introduction

Program synthesis is an automated or semi-automated process of deriving a program, i.e. generating code, from a 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 [11], examples [13] or rich types such as polymorphic refinement types [25] or graded types [17]. Regardless of the kind of specification, program synthesis must deal with two main inherent sources of complexity – search over the space of valid programs, and interpreting user intent.

Synthesis is said to be *type-driven* when it uses types as a form of program specification and produces an expression whose type matches the specification. Type-driven synthesis frameworks usually leverage rich types as a way to make specifications more expressive and prune the valid programs search space, while maintaining a "familiar" specification interface (types) for the user. Richer type systems allow for more precise types, which can statically eliminate various kinds of logical errors by making certain invalid program states ill-typed (e.g., a "null aware" type system will ensure at compile-time that you cannot dereference a null-pointer). For instance, the type $\mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$ specifies a (curried) function that takes two integers and produces an integer. Viewed as a specification, it is extremely imprecise (there are an infinite number of functions that satisfy this specification). However, the richer type $(x{:}\mathsf{Int}) \rightarrow (y{:}\mathsf{Int}) \rightarrow \{z{:}\mathsf{Int} \mid z = x + y\}$ very precisely specifies a function that takes integer arguments x and y and returns an integer z that is the sum of x and y.

The focus of our work is on type-driven synthesis where specifications take the form of linear types. Linear types constrain resource usage in programs by *statically* limiting the number of times certain resources can be used during their lifetime (linear resources must be used *exactly once*). They can be applied to resource-aware programming such as concurrent programming (e.g. session types for message passing concurrency [7]), memory-management (e.g. Rust's ownership types), safely updating-in-place mutable structures [5], enforcing protocols for external APIs [5], to name a few.

Contributions and Outline. Despite their long-known potential [28,7,5] and strong proof-theoretic foundations [4,10,8], synthesis with linear types combined with other advanced typing features has generally been overlooked in the literature. In this work we present a framework for synthesis with linear types extended with recursive algebraic data types, parametric polymorphism and refinements. We first introduce linear types as specifications and outline the synthesis process, leveraging linearity, by example (§ 2). We then discuss the formal system driving the synthesis (§ 3) and describe the architecture of our framework named *SILI*, examining technical details and key implementation challenges (§ 4). Finally, we evaluate our work through expressiveness and performance benchmarks (§ 6) and discuss related work (§ 5). Appendix A covers background concepts such as *linear logic* and *sequent calculus*. Appendix B presents the final set of inference rules. Appendix C lists concrete examples of synthesis with *SILI*.

2 Overview

The *SILI* synthesizer combines linear types with recursion, parametric polymorphism, recursive algebraic data types, and refinement types. The synthesizer is built on top of a system of proof-search for linear logic. Proof search relates to program synthesis via the Curry-Howard correspondence [12,16,29], which states that *propositions in a logic* are *types*, and *proofs* of those propositions are well-typed *programs* – finding a proof of a proposition is finding a program with that type.

Linear types make for more precise specifications than simple types because information on which resources must necessarily be used is encoded in the type. For instance the type $\mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int}$ denotes a function that *must* use its two integer arguments to produce an integer. Their preciseness also affects the search space: all programs where a linear resource is used non-linearly (i.e. not *exactly once*) are ill-typed. With linearity built into the synthesis process, usage of a linear proposition more than once is not considered, and unused propositions are identified during synthesis, constraining the space of valid programs.

The core of the synthesis is a *sound* and *complete* system consisting of *bottom-up* proof-search in propositional linear logic based on *focusing* []. Our approach, being grounded by 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). Moreover, we can leverage the modularity of the proof-search based approach along two axes: first, since proof search need not construct only closed proofs, we can effectively synthesize program sub-expressions (i.e. synthesis based on typed holes); secondly, the framework is amenable to extensions to the core propositional language, allowing for the introduction of a richer type structure while preserving the correctness of programs by construction.

The *SILI* programming language is naturally developed alongside the synthesizer. Synthesis goals are inserted in the program by use of a *synth* keyword or mark, indicating a program should be generated for a given type. Additionally, it opens the possibility of synthesis within a context, that is, with knowledge of other functions and structures defined in the same program.

Core linear calculus. Initially, we synthesize from specifications that only use literals and propositional linear types, such as linear logic theorems. Namely, the specification $(A \multimap B \multimap C) \multimap (A \otimes B) \multimap C$ produces the function `\a -> \b -> let c*d = b in a c d`, which takes an argument `a`, a *curried* linear function of type $A \multimap B \multimap C$, and an argument `b`, a *linear pair* of an A and a B and produces a value of type C by deconstructing the pair and applying `a` to the corresponding elements of the pair (i.e., *uncurries* the function `a`).

Beyond propositional logic. To be able to synthesize more interesting programs – and thus, empirically prove the feasability of more relevant synthesis in a linear context, we extend the syntax and type system with recursiveness, parametric polymorphism, algebraic data types (ADTs) and type refinements.

ADTs. We reiterate the expressiveness of linear types: by requiring some types to be used linearly (consumed), we can assure, e.g., the deconstruction of an ADT. As such, the specification that identifies a function that takes as arguments an unrestricted (!) linear function[1] that converts as in bs, a list of as, and produces a list of bs, written $!(a \multimap b) \multimap \mathit{List}\ a \multimap \mathit{List}\ b$, isn't satisfied by the program `(\x -> \y -> Nil)`. To synthesize a valid program from it, `(List a)` has to be deconstructed, and its constructor arguments used. *SILI* would output the following function (*map*) without relying on any additional information:

```
map !e d = case d of
      Nil -> Nil
    | Cons g*h -> Cons (e g, map (!e) h);
```

Refinements. Refinements are logical predicates which must hold for any term that inhabits a given type. The *SILI* language supports simple refinements on integers with arithmetic expressions in the predicates. They assert the robustness of our main framework, i.e., how it is amenable to significant additions without necessarily interfering with the overall approach. As an example, from the specification $x\{Int\} \multimap y\{Int\} \multimap z\{Int\} \multimap k\{Int \mid x + k = y * z\}$ (a function that takes three integers and produces an integer that satisfies the given predicate), the program `(\a -> \b -> \c -> ((0 - 1) * a) + (c * b))` is synthesized.

Guiding the Search. Specifications can be additionally augmented with four keywords: "using", "depth", "assert" and "choose". Through them, we can fine-tune the synthesis process and, respectively, force certain functions to be present in the synthesis outcome, allow a *deeper* search in the valid programs space, require given predicates to hold true in the program after the synthesis is complete, and select between alternative results.

Taken directly from the Linear Haskell paper [5], we present a more intricate example of synthesis: given the linear type signatures for array primitives (*newMArray* passes a new mutable array to a function that uses it linearly, *write* takes a mutable array and writes a pair to it, *freeze* consumes a mutable array and produces an imutable array, *foldl* has the default definition) we synthesize a function "array" that provides an immutable array given a list of pairs to write. The input program is formulated as follows:

[1] might be used non-linearly, but its parameter is used linearly in its body

```
foldl :: (a -o b -o a) -> a -o (List b) -o a;
newMArray :: Int -> (MArray a -o !b) -o b;
write :: MArray a -o (Int * a) -> MArray a;
freeze :: MArray a -o !(Array a);
synth array :: Int -> List (!(Int * a)) -> Array a
    | using (foldl) | depth 3;
```

Matching the linear definition for *array* from [5], *SILI* outputs (*in 0.1s*):

```
array !d !e = newMArray (!d) (\j -> freeze (foldl (!write) j e));
```

3 The *SILI* Synthesis Framework

Program synthesis from linear types with polymorphism, recursive algebraic data types, and refinements, is essentially new in the synthesis literature. Despite the substantial amount of research on linear logic and proof-search upon which we base our core synthesizer, formal guidelines for richer types and their intrinsic challenges (such as infinite recursion) must be developed.

In this section we formalize the techniques that guide synthesis from our more expressive specifications, alongside the already well defined rules that model the core of the synthesizer, putting together a *sound* set of inference rules that characterizes our framework for linear synthesis of recursive programs from specifications with the select richer types, and describes the system in enough detail for the synthesizer to be reproducible by a theory-driven implementation. We note that a *sound* set of rules guarantees we cannot synthesize incorrect programs; and that the valid programs derivable through them reflect the subjective trade-offs we committed to. Different choices and approaches outside the core might lead to completely distinct synths and spaces of valid programs.

Core Rules. The system comprises of proof search in (intuitionistic) linear logic sequent calculus, based on a system of resource management [8,18] and focusing.

The core language is a simply-typed linear λ-calculus with linear functions ($\multimap$), additive ($\&$) and multiplicative ($\otimes$) pairs (denoting alternative *vs* simultaneous occurrence of resources), multiplicative unit ($\mathbf{1}$), additive sums ($\oplus$) and the exponential modality ($!$) (to internalize unrestricted use of variables). The syntax of terms (M, N) and types (τ, σ) is given below:

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

$$\tau, \sigma \ ::= a \mid \tau \multimap \sigma \mid \tau \,\&\, \sigma \mid \tau \otimes \sigma \mid \mathbf{1} \mid \tau \oplus \sigma \mid !\tau$$

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. Andreoli's *focusing* for linear logic [4] is a technique to remove non-essential non-determinism from proof-search by structuring the application of so-called invertible and non-invertible inference rules. Andreoli 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 losing 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 are $\otimes, \mathbf{1}, \oplus, !$.

Given this separation, focusing divides proof search into two phases: the inversion phase ($\Uparrow$), in which we apply *all* invertible rules eagerly, and the focusing phase ($\Downarrow$), in which we decide a proposition to focus on, and then apply non-invertible rules, staying *in focus* until we reach an asynchronous (i.e. 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 $\Gamma; \Delta \vdash A$, meaning that A is derivable from the linear assumptions in Δ and unrestricted assumptions in Γ, is split into four judgments, grouped into the two phases ($\Uparrow, \Downarrow$). For the invertible phase, an *inversion* context Ω holds propositions that result from decomposing connectives. The right inversion and left inversion judgments are written $\Gamma; \Delta; \Omega \vdash A \Uparrow$ and $\Gamma; \Delta; \Omega \Uparrow \vdash A$, respectively, where the $\Uparrow$ indicates the connective or context being inverted. For the focusing phase (i.e. 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 $\Gamma; \Delta \vdash A \Downarrow$ and the left focus judgment is written $\Gamma; \Delta; B \Downarrow \vdash A$, where $\Downarrow$ indicates the proposition under focus.

To handle the context splitting required to prove subgoals, we augment the judgments above using Hodas and Miller's resource management technique where a pair of input/output linear contexts is used to propagate the yet unused linear resources across subgoals; e.g. the left inversion judment is written $\Gamma; \Delta/\Delta'; \Omega \Uparrow \vdash A$ where Δ is the input linear context and Δ' is the output one.

Putting together linear logic and linear lambda calculus through the Curry-Howard correspondence, resource management, and focusing, we get the following core formal system (inspired by [10,23]) – 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, synthesize a program of type B with an additional hypothesis of type A named x in the Ω context, resulting in the program M and output linear context Δ' that cannot contain the added hypothesis $x{:}A$. Finally, the resulting program is $\lambda x.M$ and the output linear context is Δ'.

We start with right invertible rules, which decompose the goal proposition until it's synchronous.

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

$$\frac{\Gamma; \Delta/\Delta'; \Omega \vdash M : A \Uparrow \qquad \Gamma; \Delta/\Delta''; \Omega \vdash N : B \Uparrow \qquad \Delta' = \Delta''}{\Gamma; \Delta/\Delta'; \Omega \vdash (M \mathbin{\&} N) : A \mathbin{\&} B \Uparrow} \ (\&R)$$

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:

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

We follow with left invertible rules for asynchronous connectives, which decompose asynchronous propositions in Ω.

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

$$\frac{\Gamma; \Delta/\Delta'; \Omega \Uparrow \vdash M : C}{\Gamma; \Delta/\Delta'; \Omega, x{:}1 \Uparrow \vdash \text{ let } \star = x \text{ in } M : C} \ (1L)$$

$$\frac{\begin{array}{l}\Gamma; \Delta/\Delta'; \Omega, y{:}A \Uparrow \vdash M : C \qquad y \notin \Delta' \\ \Gamma; \Delta/\Delta''; \Omega, z{:}B \Uparrow \vdash N : C \qquad z \notin \Delta'' \qquad \Delta' = \Delta''\end{array}}{\Gamma; \Delta/\Delta'; \Omega, x{:}A \oplus B \Uparrow \vdash \text{ case } x \text{ of inl } y \rightarrow M \mid \text{inr } z \rightarrow N : C} \ (\oplus L)$$

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

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:

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

After inverting all the asynchronous propositions in Ω we'll 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:

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

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

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 $!R$).

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

$$\frac{\Gamma; \Delta/\Delta'; y{:}A \Downarrow \vdash M : C}{\Gamma; \Delta/\Delta'; x{:}A \,\&\, B \Downarrow \vdash M\{(\text{fst } x)/y\} : C}\ (\&L_1)$$

$$\frac{\Gamma; \Delta/\Delta'; y{:}B \Downarrow \vdash M : C}{\Gamma; \Delta/\Delta'; x{:}A \,\&\, B \Downarrow \vdash M\{(\text{snd } x)/y\} : C}\ (\&L_2)$$

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

$$\frac{\Gamma; \Delta/\Delta' \vdash M : A \Downarrow}{\Gamma; \Delta/\Delta' \vdash \text{ inl } M : A \oplus B \Downarrow}\ (\oplus R_1) \qquad \frac{\Gamma; \Delta/\Delta' \vdash M : B \Downarrow}{\Gamma; \Delta/\Delta' \vdash \text{ inr } M : A \oplus B \Downarrow}\ (\oplus R_2)$$

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

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:

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

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

The rules written above together make the core of our synthesizer. Next, we'll present new rules that align and build on top of these to synthesize recursive programs from more expressive (richer) types.

Algebraic Data Types. 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, which might be, e.g., a product type ($A \otimes B$), or unit (1), in practice allowing for constructors with an arbitrary number of parameters. In the *SILI* language, the programmer can define custom ADTs; as an example, we show the definition of an ADT which holds zero, one, or two values of type A, using the syntax: `data Container = None 1 | One A | Two (A * A)`. The grammar is extended as (where C is an ADT constructor and T is an ADT):

$$\begin{array}{ll} M, N & ::= \dots \mid C_n\ M \mid (\mathsf{case}\ M\ \mathsf{of}\ \dots \mid C_n\ u \Rightarrow N) \\ \tau, \sigma & ::= \dots \mid T \end{array}$$

The semantics of ADTs relate to those of the plus ($\oplus$) type – both are additive disjunctions. To construct a value of an ADT we must use one of its constructors, similar to the way $\oplus$ requires only proof of either the left or right type it consists of. Analogously, we can deconstruct a value of an ADT via pattern matching on its constructors, where all branches of the pattern match must have the same type – akin to the left rule for the $\oplus$ connective. In effect, the inference rules for a simple ADT are a generalized form of the $\oplus$ rules. Therefore, there's one left rule for ADTs, and an arbitrary number of right rules, one for each constructor, where ADT T and its constructors stand for any ADT defined as `data T = C1 X1 | C2 X2 | ... | Cn Xn`:

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

$$\frac{\begin{array}{ll} \Gamma; \Delta/\Delta'_1; \Omega, y_1{:}X_1 \Uparrow \vdash M_1 : C & y_1 \notin \Delta'_1 \\ \Gamma; \Delta/\Delta'_2; \Omega, y_2{:}X_2 \Uparrow \vdash M_2 : C & y_2 \notin \Delta'_2 \\ \quad \dots & \\ \Gamma; \Delta/\Delta'_n; \Omega, y_n{:}X_n \Uparrow \vdash M_n : C & y_n \notin \Delta'_n \qquad \Delta'_1 = \Delta'_2 = \dots = \Delta'_n \end{array}}{\Gamma; \Delta/\Delta'_1; \Omega, x{:}T \Uparrow \vdash\ \mathsf{case}\ x\ \mathsf{of}\ \dots \mid C_n\ y_n \rightarrow M_n : C}\ (\textsc{adtL})$$

A more general formulation of ADTs says an ADT can be recursive (or "inductively defined"), meaning constructors can take as arguments values of the type they are defining. This change has a significant impact in the synthesis process. Take, for instance, the ADT defined as `data T = C1 T`, the synthesis goal $T \multimap C$, and part of its derivation:

$$\frac{\dfrac{\dfrac{\dots}{\Gamma; \Delta/\Delta'; \Omega, y{:}T \Uparrow \vdash \mathsf{case}\ y\ \mathsf{of}\ C_1\ z \rightarrow \dots : C}\ (\textsc{adtL})}{\Gamma; \Delta/\Delta'; \Omega, x{:}T \Uparrow \vdash \mathsf{case}\ x\ \mathsf{of}\ C_1\ y \rightarrow \dots : C}\ (\textsc{adtL})}{\Gamma; \Delta/\Delta'; \Omega, x{:}T \vdash \dots : C \Uparrow}\ (\Uparrow R)$$

Using our current system, we are to apply an infinite number of times (adtL), never closing the proof. Symmetrically, the derivation for goal T is also infinite.

$$(\textsc{adtR})\ \frac{(\textsc{adtR})\ \dfrac{(\textsc{adtR})\ \dfrac{\dots}{\Gamma; \Delta/\Delta'; \Omega \vdash C_1 \cdots : T \Downarrow}}{\Gamma; \Delta/\Delta'; \Omega \vdash C_1 \cdots : T \Downarrow}}{\Gamma; \Delta/\Delta'; \Omega \vdash C_1 \cdots : T \Downarrow}$$

To account for this situation, we impede the decomposition of an ADT in subsequent proofs of its branches, and, symmetrically, don't allow construction of an ADT when trying to synthesize an argument for its constructor. For this, we need two more contexts, P_C for constraints on construction and P_D for constraints on deconstruction. Together, they hold a list of ADTs that cannot be constructed or deconstructed at a given point in the proof. For convenience, they are represented by a single P if unused. All non-ADT rules trivially propagate these. The ADT rules are then extended as follows, where $P'_C = P_C, T$ if T is recursive and $P'_C = P_C$ otherwise (P'_D is dual):

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

$$\frac{\begin{array}{l} T \notin P_D \qquad \Delta'_1 = \cdots = \Delta'_n \\ (P_C; P'_D); \Gamma; \Delta/\Delta'_1; \Omega, y_1{:}X_1 \Uparrow \vdash M_1 : C \qquad y_1 \notin \Delta'_1 \\ \quad \ldots \\ (P_C; P'_D); \Gamma; \Delta/\Delta'_n; \Omega, y_n{:}X_n \Uparrow \vdash M_n : C \qquad y_n \notin \Delta'_n \end{array}}{(P_C; P_D); \Gamma; \Delta/\Delta'_1; \Omega, x{:}T \Uparrow \vdash\ \text{case } x \text{ of } \ldots \mid C_n\ y_n \rightarrow M_n : C}\ (\textsc{adt}\mathrm{L})$$

These modifications prevent the infinite derivations in the scenarios 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 three rules to complement the restrictions on construction and destruction of recursive types. First, since we can't deconstruct some ADTs any further because of a restriction, 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 isn't restricted. The three following rules ensure this:

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

$$\frac{}{P; \Gamma; \Delta/\Delta'; x{:}T \Downarrow \vdash x : T}\ (\textsc{adt-init})$$

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

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.

Recursion. The main idea behind synthesis of recursive programs is the labeling of the main goal and the addition of its type, under that name, to the unrestricted context. That is, to synthesize a recursive function of type $A \multimap B$ named f, the initial judgment can be written as

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

and, by definition, all subsequent inference rules will have $(f{:}A \multimap B)$ in the Γ context too. We can also force the usage of the recursive call by adding it not only to the unrestricted context, but to the linear one as well. 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 via the following invariant: the recursive hypothesis can only be used in *recursive branches of ADT deconstruction*, i.e. the recursive call should only take "smaller", recursive, hypothesis as arguments. To illustrate, in any recursive function with a list argument (whose type is defined as `data List = Nil | Cons (A * List)`), recursive calls are only allowed when considering a judgment of the form List $\vdash C$, i.e. when a list value is available to produce the goal C, and only in the *Cons* branch. Furthermore, we also forbid the usage of the recursive function when synthesizing arguments to use it.

Polymorphic Types. A polymorphic specification is a type of form $\forall\overline{\alpha}.\ \tau$ where $\overline{\alpha}$ is a set of variables that stand for any (non-polymorphic) type in τ. Such a type is also called a *scheme*. Synthesis of a scheme comprises of turning it into a non-quantified type, 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 inference method's [21,15] 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 $\alpha0 \multimap \alpha0$. We add a rule for this, where $\forall\overline{\alpha}.\ \tau \sqsubseteq \tau'$ indicates type τ' is an *instantiation* of type scheme $\forall\overline{\alpha}.\ \tau$:

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

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 (nothing constrains their type), i.e. a polymorphic program. The simplest example is the polymorphic function *id* of type $\forall\alpha.\ \alpha \multimap \alpha$. The program synthesized from that specification is $\lambda x.x$, a lambda abstraction that does not constrain the type of its parameter x in any way.

The main challenge of polymorphism in synthesis is the usage of schemes from the unrestricted context. To begin with, 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 the same way the linear context is (by having an input and output context (Θ/Θ')). In contrast to Hindley-Milner's inference, everytime a constraint is added it is solved against all other constraints – a branch of the proof search is desired to fail as soon as possible. Note that we instantiate the scheme with *existential* type variables ($?\alpha$) rather than just 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 assigned a 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 wether constrain c can be unified with those in Θ:

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

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

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

Further Challenges. We now consider two more sources of infinite recursion in the synthesis process. The first is the use of an unrestricted function to synthesize a term of type τ that in turn will require a term of the same type τ. An example is the sub-goal judgment $(a \multimap b \multimap b); (a \multimap b \multimap b) \Downarrow\vdash b$ that appears while synthesizing *foldr* – we apply ($\multimap$L) until we can use INIT ($b \Downarrow\vdash b$), and then we must synthesize an argument of type b. Without any additional restrictions, we may become again left focused on $(a \multimap b \multimap b)$, and again require b, and on and on. The solution will be to disallow the usage of the same function to synthesize the same goal a second time further down in the derivation.

The other situation occurs when using an unrestricted polymorphic function that requires synthesis of a term with an existential type when the goal is an existential type. In contrast to the previous problem, the type of the goal and of the argument that will cause the loop won't match exactly, since instantiated bound variables are always fresh. For example, for $\forall\alpha, \beta.\ \alpha \multimap \beta \multimap \beta; ?\alpha \multimap ?\beta \multimap ?\beta \Downarrow\vdash ?\sigma$, we'll unify $?\beta$ with $?\sigma$, and then require a term of type $?\beta$ (not

$?\sigma$). We want to forbid the usage of the *same* function to attain *any* existential goal, provided that function might create existential sub-goals (i.e. it's polymorphic). However, we noticed that, even though for most tried problems this "same function" approach worked, context-heavy problems such as *array* (seen in § 2) wouldn't terminate in a reasonable amount of time. As such, we'll instead define that, given an existential[2] goal C, we can only "decide left!" on a proposition A if, altogether, the amount of times we've "decided left!" on an polymorphic function to produce an existential goal is less than a *constant "existential depth"* d_e (which controls a *depth* aspect of the synthesis process).

Extending the restrictions context (P) with restrictions on using the unrestricted context ($P_{L!}$), we modify DECIDELEFT! to formalize the two previous paragraphs, where $\textsc{isExist}(C)$ is true if C is an existential type, $\textsc{isPoly}(f)$ is true if f is universally quantified (i.e. f has form $\forall\overline{\alpha}f'$), and $P'_{L!} = P_{L!}, (A, C)$ if A is a function and $P'_{L!} = P_{L!}$ otherwise:

$$\frac{\begin{array}{l}(A, C) \notin P_{L!} \\ \textsc{isExist}(C) \Rightarrow |\{u \mid (f, u) \in P_{L!}, \textsc{isPoly}(f), \textsc{isExist}(u)\}| < d_e \\ \Theta/\Theta'; (P_C, P_D, P'_{L!}); \Gamma, A; \Delta/\Delta'; A \Downarrow \vdash C\end{array}}{\Theta/\Theta'; (P_C, P_D, P_{L!}); \Gamma, A; \Delta/\Delta'; \cdot \Uparrow \vdash C} \ (\textsc{decideL!})$$

Polymorphic ADTs. To allow type parameters and the use of universally quantified type variables in ADT constructors, we must guarantee that the ADT-INIT rule can unify the type parameters and that when constructing or destructing an ADT, type variables in constructor parameters are substituted by the actual type (i.e. to construct `List Int` with `data List a = Cons (a * List a)`, we wouldn't try to synthesize `(a * List a)`, but rather `(Int * List Int)`). To unify $T_{\overline{\alpha}}$ with $T_{\overline{\beta}}$, the sets of type parameters $\overline{\alpha}$ and $\overline{\beta}$ must satisfy $|\overline{\alpha}| = |\overline{\beta}|$ together with $\forall i \ 0 \leq i \land i < |\overline{\alpha}| \land \textsc{unify}(\overline{\alpha}_i \mapsto \overline{\beta}_i)$. The constructor type substitution needn't be explicit in the rule:

$$\frac{\textsc{unify}(T_{\overline{\alpha}} \mapsto T_{\overline{\beta}}, \Theta)}{\Theta/\Theta, T_{\overline{\alpha}} \mapsto T_{\overline{\beta}}, P; \Gamma; \Delta/\Delta'; x{:}T_{\overline{\alpha}} \Downarrow \vdash x : T_{\overline{\beta}}} \ (\textsc{adt-init})$$

Refinement Types. Refinement types are types with a predicate (a non-existing predicate is the same as it being *true*); dependent types are functions with refinement types in which the argument type is labeled and said label can be used in the predicates of the return type (e.g. $(x : \mathsf{Int}) \multimap \{y : \mathsf{Int} \mid y = x\}$ specifies a function that takes an Int and returns an Int of equal value). We extend the types syntax with our refinement types:

$$\begin{array}{lcl}\tau & ::= & \ldots \mid (x : \tau) \multimap \sigma \mid \{x : \tau \mid P\} \\ P & ::= & P = P \mid P \neq P \mid P \vee P \mid P \wedge P \mid P \Rightarrow P \mid n = n \mid n \neq n \\ & & \mid n \leq n \mid n \geq n \mid n < n \mid n > n \mid \mathit{true} \mid \mathit{false} \mid x \\ n & ::= & n * n \mid n + n \mid n - n \mid \langle \mathit{natural} \rangle \mid x\end{array}$$

[2] a type is existential when any of its components is an existential type variable

The addition of refinement types to the synthesizer doesn't interfere with the rest of the process. We define the following right and left rule, to synthesize or consume in synthesis a refinement type, where $\textsc{getModel}(p)$ is a call to an SMT solver that returns a model of an uninterpreted function that satisfies $\forall_{a,b,\ldots,n}\ h_a \Rightarrow h_b \Rightarrow \cdots \Rightarrow h_n \Rightarrow p$, where n is the refinement type label and h_n its predicate, with $a, \ldots, n$ standing for the label of every refinement type in the propositional contexts; and $\textsc{sat}(p_a \Rightarrow p_b)$ is a call to an SMT solver that determines universal satisfiability of the implication between predicates (the left focused proposition subtypes the goal).

$$\frac{\textsc{getModel}(p) = M}{\Theta/\Theta'; P; \Gamma; \Delta/\Delta' \vdash M : \{a : A \mid p\} \Uparrow}\ (\textsc{ref}\mathrm{R})$$

$$\frac{\textsc{sat}(p_a \Rightarrow p_b)}{\Theta/\Theta'; P; \Gamma; \Delta/\Delta'; x{:}(\{a : A \mid p_a\}) \Downarrow \vdash x : \{b : A \mid p_b\}}\ (\textsc{ref}\mathrm{L})$$

Optimizations. To speed up the process and get a cleaner output, we add a rule that lets us "skip some rules" if left focused on a !-ed proposition, and the goal is !-ed:

$$\frac{\Theta/\Theta'; P; \Gamma; \Delta; x{:}A \Downarrow \vdash M : C}{\Theta/\Theta'; P; \Gamma; \Delta; x{:}!A \Downarrow \vdash M : !C}\ (! \Downarrow\mathrm{L})$$

4 Architecture

The *SILI* synthesizer operates as part of the pipeline that processes a full program in the *SILI* language, and is called to generate terms marked (by the syntax {{ ... }}) for synthesis. The main pipeline consists of:

$$\text{Parsing} \rightarrow \text{Desugaring} \rightarrow \text{Inference} \rightarrow \text{Synthesis} \rightarrow \text{Evaluation}$$

The parsing module converts ADT declarations and top level functions written in the language's syntax to a list of abstract syntax trees (ASTs) understood by the rest of the pipeline.

The desugaring module converts the frontend AST to a core AST without syntatic sugar and using the locally-nameless representation [9], in which all bound variables are represented with Debruijn indices rather than names (i.e. using the distance to the binding site in terms of the number of traversed binders), This way inference can be done without worrying about name conflicts.

The inference module will traverse the list of ASTs in order (top-down), and infer the type of all functions defined by the user, checking the inferred type against a possible user given type annotation. Whenever a synthesis mark is found in the AST, all the inferrence context up to that point is added to the mark, and, if not explicit, its type is inferred, essentially defining the synthesis context and synthesis goal.

The synthesis module iterates over the marks and runs the synthesizer with the respective context and goal for each mark. The resulting expression is checked against the synthesis constraints defined through some keywords, continuing synthesis if they are violated: “using” guarantees that certain functions are used in the program body (to invalidate valid program without them), and “assert” evaluates the given assertion within the synthesized program to a validating boolean. Additionally, the keyword “depth” controls the *existential depth* (seen in § 3), and “choose” selects a different result as long as one is available. Finally, the expression is minimized (i.e. η-reduced), and the mark in the frontend AST is replaced with the synthesized program.

The evaluation module evaluates the function named “main” within the complete synthesized program, returning a final value. When evaluating synthesis assertions, a timeout is used to interrupt possibly non-terminating programs.

4.1 Implementation

The implementation was carried out in Haskell. We highlight some key points. The project and source code are available from the repository [20].

Backtracking. Although focusing reduces non-determinism during proof search, it cannot fully eliminate it. At several points during proof construction we must choose one of multiple rules to apply, and in the event of not being able to construct the proof following that choice, we want to backtrack to the decision point and attempt to derive a proof taking a different route. To this effect, our prototype synthesizer depends on the Haskell library *LogicT*, "a backtracking logic-programming monad” [2].

SMT Solving. To typecheck and synthesize refinement types and dependent functions, we interface with an SMT solver to check satisfiability. For this, we use the library *SBV* [3]. Unfortunately, this library does not support first-order logic with uninterpreted functions. Since this is needed for synthesis with refinement types when right focused, we instead use unsafe library primitives to communicate with the solver directly, constructing the logical formulas directly. For satisfying formulas, for instance, when typechecking, or left focused on a refinement type, we use the library API as an oracle.

Memoization. Some proofs attempt to synthesize the same sub-goal with the same premises more than once. Since these proof attempts might be expensive performance-wise, and we can rely on the program’s determinism (equal calls to synthesize result in equal outcomes), we added memoization to our synthesizer – an optimization technique that stores function calls and returns cached values for equal inputs. For our synthesizer, two calls are equivalent when contexts $\Theta, P, \Gamma, \Delta$, Ω, (or alternatively the left focus), and the goal, are equivalent. Because comparing all of these is slow, our implementation hashes the relevant values via a hashing library (*Hashable* [1]) and uses the result as the key. Furthermore, a branch we want to memoize might fail, not providing any result

– we also want to record synthesis failure, as it happens more often than successful results. We ran benchmarks to measure the impact of memoization on performance. It's quite noticeable in synthesis with multiple functions in the unrestricted context, or more complex specifications. For example, if we add a new function *read* to the unrestricted context in the *array* problem seen in § 2, synthesis with memoization will take around $9s$, while synthesis without memoization doesn't terminate after $15m$. The implementation could be greatly improved, but we leave such optimizations to future work.

Debugging. The rules described in the formal system (§ 3) to handle infinite recursion might seem obvious in retrospective, however, when faced with a program that doesn't terminate, what went wrong is not so clear. To make the debugging experience better, we developed a tracing system that prints information whenever a rule is applied, alongside the stack of rules applied so far.

Interface. A simple, syntax-highlighted, web interface was developed alongside a server to make possible experimenting with the *SILI* language without having to download and compile the complete toolchain. The web interface is available from the repository [20].

Example. To showcase the expressiveness of our synthesizer, we present as examples the input and output program for *reverse*. Note how we can guide the synthesis using the assertion mechanism to supply an example:

```
synth reverse :: List a -o List a -o List a
  | assert (reverse (Cons (1, Cons (2, Nil))) Nil) == (Cons (2, Cons (1, Nil)));

reverse b c = case b of
    Nil -> c
  | Cons e*f -> reverse f (Cons (e, c));
```

5 Related Work

Type-based program synthesis is a vast field of study and so it follows that a lot of literature is available to inspire and complement our work. Most works [17,25,22,13] follow some variation of the synthesis-as-proof-search approach. However, the process is novel for each due to a variety of different rich types explored and their corresponding logics and languages; or nuances of the synthesis process itself, such as complementing types with program examples; or even the programming paradigm of the output produced (e.g. generating heap manipulating programs [26]).

Program Synthesis from Polymorphic Refinement Types. The work [25] also studies synthesis of recursive functional programs in an "advanced" context. Their specifications combine two rich forms of types: polymorphic and refinement types. Their approach to refinement types consists of a new algorithm that supports decomposition of the refinement specification. We also support refinements (and polymorphism), but they are not as integrated in the synthesis process as in [25]. Instead, our synthesizer leverages the expressiveness of linear types and techniques for proof-search in linear logic to guide its process.

Resourceful Program Synthesis from Graded Linear Types. The work [17] 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 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 more complex, and so they provide solutions which adapt Hodas and Miller's approach [8,18]. 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 [7,6], which is a crucial avenue of future work. Moreover, their use of grading effectively requires constraint solving 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.

6 Evaluation

The benchmarks[3] are separated into groups: theorems of linear logic, recursive ADTs, refinements. The first table displays successful synthesis – for each goal we analyze the mean time and standard deviation, the conditions used to guide the output, and the components (other functions) used in the resulting program.

Group	Goal	Avg. time $\pm\,\sigma$	Keywords	Components
Linear Logic Theor.	uncurry	$133\mu s \pm 4.9\mu s$		
	distributivity	$179\mu s \pm 5.0\mu s$		
	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	$505\mu s \pm 18\mu s$		
	uncons	$215\mu s \pm 15\mu s$		
	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			*depth 3*	*freeze, foldl*
	array 2	$80ms \pm 870\mu s$	*using (foldl),depth 3*	*newMArray,write*
Refinements	add3	$39ms \pm 1.1ms$		+

[3] ran on an Intel Core i5 machine with 8GB of RAM using the benchmarking library Criterion, and considering the full pipeline pass, from parsing to synthesis

References

1. hashable: A class for types that can be converted to a hash value. https://hackage.haskell.org/package/hashable
2. logict: A backtracking logic-programming monad. https://hackage.haskell.org/package/logict
3. sbv: Smt based verification: Symbolic haskell theorem prover using smt solving. https://hackage.haskell.org/package/sbv
4. 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
5. 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
6. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR 2010. pp. 222–236 (2010)
7. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
8. 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
9. Chargu'eraud, A.: The locally nameless representation. J. Autom. Reason. **49**(3), 363–408 (2012). https://doi.org/10.1007/s10817-011-9225-2, https://doi.org/10.1007/s10817-011-9225-2
10. 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
11. 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)
12. Curry, H.: Functionality in combinatory logic. vol. 20, pp. 584–590. Department of Mathematics, The Pennsylvania State College (1934). https://doi.org/10.1073/pnas.20.11.584
13. 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

14. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
15. 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`
16. Howard, W.A.: The formulae-as-types notion of construction. pp. 479–490 (1980 (originally circulated 1969))
17. 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`
18. 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`
19. Liang, C., Miller, D.: A unified sequent calculus for focused proofs. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA. pp. 355–364 (2009). https://doi.org/10.1109/LICS.2009.47, `https://doi.org/10.1109/LICS.2009.47`
20. Mesquita, R., Toninho, B.: The SILI synthesizer. `https://github.com/alt-romes/slfl` (July 2021)
21. 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`
22. 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`
23. Pfenning, F.: Focus handouts. `https://www.cs.cmu.edu/~fp/courses/15816-f01/handouts/focus.pdf`
24. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
25. 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`
26. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. PACMPL **3**(POPL), 72:1–72:30 (2019). https://doi.org/10.1145/3290385, `https://doi.org/10.1145/3290385`
27. Prawitz, D.: Natural Deduction. Almquist & Wiksell (1965)
28. Wadler, P.: Linear types can change the world! In: PROGRAMMING CONCEPTS AND METHODS. North (1990)
29. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). https://doi.org/10.1145/2699407, `https://doi.org/10.1145/2699407`

A Background

Type Systems. A type system can be formally described through a set of inference rules that inductively define a judgment of the form $\Gamma \vdash M : A$, stating that program expression M has type A according to the *typing assumptions* for variables tracked in Γ. For instance, $x{:}\mathsf{Int}, y{:}\mathsf{Int} \vdash x + y : \mathsf{Int}$ states that $x+y$ has type Int under the assumption that x and y have type Int. An expression M is deemed well-typed with a given type A if one can construct a typing derivation with $M : A$ as its conclusion, by repeated application of the inference rules.

The simply-typed λ-calculus is a typed core functional language [24] that captures the essence of a type system in a simple and familiar environment. Its syntax consists of functional abstraction, written $\lambda x{:}A.M$, denoting an (anonymous) function that takes an argument of type A, bound to x in M; and application $M\,N$, with the standard meaning, and variables x. For instance, the term $\lambda x{:}A.x$, denoting the identity function, is a functional abstraction taking an argument of type A and returning it back.

Propositions as Types. It turns out that the inference rules of the simply-typed λ-calculus are closely related to those of a system of natural deduction for intuitionistic logic [27]. This relationship, known as the Curry-Howard correspondence (see [29] for a historical survey), identifies that the propositions of intuitionistic logic can be read as types for the simply-typed λ-calculus ("propositions as types"), their proofs are exactly the program with the given type ("proofs as programs"), and checking a proof is type checking a program ("proof checking as type checking").

Inference rules in natural deduction are categorized as introduction or elimination rules; these correspond, respectively, to rules for constructors and destructors in programming languages (e.g. function abstraction vs application, construction of a pair vs projection).

We introduce select typing rules to both show how a type system can be formalized and to show the relationship with (intuitionistic) propositional logic. The following rule captures the nature of a hypothetical judgment, allowing for reasoning from assumptions:

$$\frac{}{\Gamma, u : A \vdash u : A}\ (\textsc{var})$$

When seen as a typing rule for the λ-calculus, it corresponds to the rule for typing variables – variable u has type A if the typing environment contains a variable u of type A.

As another concrete example, let us consider the rule for implication ($A \to B$):

$$\frac{\Gamma, u : A \vdash M : B}{\Gamma \vdash \lambda u.M : A \to B}\ (\to I)$$

Logically, the rule states that to prove $A \to B$, we assume A and prove B. Through the Curry-Howard correspondence, implication corresponds to the function type – the program $\lambda u.M$ has type $A \to B$, provided M has type B under the assumption that u is a variable of type A. The rule shown above is an *introduction* rule, that introduces the "implication" connective.

Finally, let us consider an *elimination* rule, also for implication:

$$\frac{\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}\ (\rightarrow E)$$

Elimination rules are easier to think of in a top-down manner. Logically, this rule states that if we prove $A \rightarrow B$ and A we can prove B. Through the Curry-Howard correspondence, implication elimination corresponds to function application, and its type is the function return type – the application MN has type B, provided M has type $A \rightarrow B$ and N has type A.

The Curry-Howard correspondence generalizes beyond the simply-typed λ-calculus and propositional intuitionistic logic. It extends to the realms of polymorphism (second-order logic), dependent types (first-order logic), and various other extensions of natural deduction and simply-typed λ-calculus – which include linear logic and linear types.

Linear Logic. Linear logic [14] can be seen as a resource-aware logic, where 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*. This usage restriction gives rise to new logical connectives, based on the way the ambient resources are used. For instance, conjunction, usually written as $A \wedge B$, appears in two forms in linear logic: multiplicative or simultaneous conjunction (written $A \otimes B$); and additive or alternative conjunction (written $A \,\&\, B$). Multiplicative conjunction denotes the simultaneous availability of resources A and B, requiring both of them to be used. Alternative conjunction denotes the availability of A and B, but where only one of the two resources may be used. Similarly, implication becomes linear implication, written $A \multimap B$, denoting a resource that will consume (exactly one) resource A to produce a resource B.

To present the formalization of this logic, besides the new connectives, we need to introduce the *resource-aware context* Δ. In contrast to the previously seen Γ, Δ is also a list of variables and their types, but where each and every variable must be used exactly once during inference. So, to introduce the connective $\otimes$ which defines a multiplicative pair of propositions, we must use exactly all the resources (Δ_1, Δ_2) needed to realize the (Δ_1) proposition A, and (Δ_2) proposition B:

$$\frac{\Delta_1 \vdash M : A \qquad \Delta_2 \vdash N : B}{\Delta_1, \Delta_2 \vdash (M \otimes N) : A \otimes B}\ (\otimes I)$$

Out of the logical connectives, we need to mention one more, since it augments the form of the judgment and it's the one that ensures logical strength i.e. we're able to translate intuitionistic logic into linear logic. The proposition $!A$ (read *of course* A) is used (under certain conditions) to make a resource "infinite" i.e.

to make it useable an arbitrary number of times. To distinguish the "infinite" variables, a separate, unrestricted, context is used – Γ. So Γ holds the "infinite" resources, and Δ the resources that can only be used once. The linear typing judgment for the introduction of the exponential $!A$ takes the form:

$$\frac{\Gamma; \emptyset \vdash M : A}{\Gamma; \emptyset \vdash !M : !A} \; (!I)$$

Logically, a proof of $!A$ cannot use linear resources since $!A$ denotes an unbounded (potentially 0) number of copies of A. Proofs of $!A$ may use other unrestricted or exponential resources, tracked by context Γ. From a computational perspective, the type $!A$ internalizes the simply-typed λ-calculus in the linear λ-calculus.

The elimination form for the exponential, written $\mathsf{let}\ !u = M\ \mathsf{in}\ N$, warrants the use of resource A an unbounded number of times in N via the variable u:

$$\frac{\Gamma; \Delta_1 \vdash M : !A \qquad \Gamma, u{:}A; \Delta_2 \vdash N : C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{let}\ !u = M\ \mathsf{in}\ N : C} \; (!E)$$

Again, through the Curry-Howard correspondence, we can view the process of finding a proof of a proposition in linear logic as the process of synthesizing a linear functional program of the given type.

Sequent Calculus. Inference rules in natural deduction (the ones we have considered so far) are ill-suited for bottom-up proof-search since elimination rules work top-down and introduction rules work bottom-up. A more suited candidate is the equivalent *sequent calculus* system in which *all* inference rules can be understood naturally in a *bottom-up* manner. The introduction and elimination rules from natural deduction disappear, and their place is taken by right and left rules, respectively. Right rules correspond exactly to the introduction rules of natural deduction, which were already understood bottom-up. The inference rule for implication introduction ($\supset$ is used instead of $\rightarrow$), seen above in natural deduction under the *propositions as types* subsection, corresponds to the following right rule in sequent calculus:

$$\frac{\Gamma, u : A \vdash M : B}{\Gamma \vdash \lambda u.M : A \supset B} \; (\supset R)$$

Intuitively, left rules act as the elimination rules of natural deduction, but are altered to work bottom-up, instead of top-down. The inference rule for the also seen above implication elimination is as follows:

$$\frac{\Gamma, u : B \vdash M : C \qquad \Gamma, u{:}A \supset B \vdash N : A}{\Gamma, u{:}A \supset B \vdash M\{(u\,N)/u\} : C} \; (\supset L)$$

As implied by their name, left rules define how to decompose or make use of a connective on the left of the turnstile $\vdash$. To use an assumption of $A \supset B$ while attempting to show some proposition C we produce a proof of A, which allows us to use an assumption of B to prove C. In terms of the corresponding λ-terms, the $\supset L$ rule corresponds to applying the variable u to the argument N but potentially deep in the structure of M.

We'll define our inference rules for synthesis under the sequent calculus system, for it simplifies the work to be done in the synthesizer.

Resource-Management. Be it from the perspective of proof *checking* (i.e. type checking) or proof *search* linear logic poses a key challenge when compared to the non-linear setting: When constructing a derivation (bottom-up), we are seemingly forced to guess how to correctly split the linear context such that the sub-derivations have access to the correct resources (e.g. the $\otimes I$ rule above).

To solve this issue we will adopt the resource-management approach of [8,19], which generalizes the judgment from $\Delta \vdash M : A$ to $\Delta_I\ \Delta_O \vdash M : A$, where Δ_I is an input context and Δ_O is an output context. Instead of requiring non-deterministic guesses of resource splits during proof search, we track which resources are used and which are remaining via the two contexts, leading to the following general strategy: to prove $A \otimes B$ using (input) resources Δ, prove A with input context Δ, consuming some subset of Δ, and produce as output the leftover resources Δ'; prove B using Δ' as its input context and then output the remaining resources Δ''; finally, after having proven $A \otimes B$, output Δ'', for subsequent derivations.

Focusing. Even with everything mentioned so far, non-determinism is still very present in proof-search, e.g. at any given point, many proof rules are applicable in general. The technique of focusing [4,10] has been previously studied as a way to discipline proof search in linear logic – a method created to trim down the search space of valid proofs in linear logic, by eagerly applying invertible rules (i.e. rules whose conclusion implies the premises), and then by "focusing" on a single connective when no more direct (invertible) rules can be applied, that is, only applying rules that breakdown the connective under focus or its subformulas. If the search is not successful, the procedure backtracks and another connective is chosen as the focus.

Focusing eliminates all of the "don't care" non-determinism from proof search, since the order in which invertible rules are applied does not affect the outcome of the search, leaving only the non-determinism that pertains to unknowns (or "don't know" non-determinism), identifying precisely the points at which backtracking is necessary.

B Formal System

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

$$\frac{\Gamma; \Delta/\Delta'; \Omega \vdash M : A \Uparrow \qquad \Gamma; \Delta/\Delta''; \Omega \vdash N : B \Uparrow \qquad \Delta' = \Delta''}{\Gamma; \Delta/\Delta'; \Omega \vdash (M \mathbin{\&} N) : A \mathbin{\&} B \Uparrow}\ (\&R)$$

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

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

$$\frac{\Gamma; \Delta/\Delta'; \Omega \Uparrow \vdash M : C}{\Gamma; \Delta/\Delta'; \Omega, x{:}1 \Uparrow \vdash \text{ let } \star = x \text{ in } M : C}\;(1L)$$

$$\frac{\begin{array}{l}\Gamma; \Delta/\Delta'; \Omega, y{:}A \Uparrow \vdash M : C \qquad y \notin \Delta' \\ \Gamma; \Delta/\Delta''; \Omega, z{:}B \Uparrow \vdash N : C \qquad z \notin \Delta'' \qquad \Delta' = \Delta''\end{array}}{\Gamma; \Delta/\Delta'; \Omega, x{:}A \oplus B \Uparrow \vdash \text{ case } x \text{ of inl } y \to M \mid \text{inr } z \to N : C}\;(\oplus L)$$

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

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

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

$$\frac{\begin{array}{l}(A, C) \notin P_{L!} \\ \textsc{isExist}(C) \Rightarrow |\{u \mid (f, u) \in P_{L!}, \textsc{isPoly}(f), \textsc{isExist}(u)\}| < d_e \\ \Theta/\Theta'; (P_C, P_D, P'_{L!}); \Gamma, A; \Delta/\Delta'; A \Downarrow \vdash C\end{array}}{\Theta/\Theta'; (P_C, P_D, P_{L!}); \Gamma, A; \Delta/\Delta'; \cdot \Uparrow \vdash C}\;(\textsc{decide}\mathrm{L}!)$$

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

$$\frac{\Gamma; \Delta/\Delta'; y{:}A \Downarrow \vdash M : C}{\Gamma; \Delta/\Delta'; x{:}A \,\&\, B \Downarrow \vdash M\{(\text{fst } x)/y\} : C}\;(\& L_1)$$

$$\frac{\Gamma; \Delta/\Delta'; y{:}B \Downarrow \vdash M : C}{\Gamma; \Delta/\Delta'; x{:}A \,\&\, B \Downarrow \vdash M\{(\text{snd } x)/y\} : C}\;(\& L_2)$$

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

$$\frac{\Gamma; \Delta/\Delta' \vdash M : A \Downarrow}{\Gamma; \Delta/\Delta' \vdash \text{ inl } M : A \oplus B \Downarrow}\;(\oplus R_1) \qquad \frac{\Gamma; \Delta/\Delta' \vdash M : B \Downarrow}{\Gamma; \Delta/\Delta' \vdash \text{ inr } M : A \oplus B \Downarrow}\;(\oplus R_2)$$

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

$$\frac{\Gamma; \Delta/\Delta'; \cdot \vdash A \Uparrow}{\Gamma; \Delta/\Delta' \vdash A \Downarrow}\ (\Downarrow R)$$

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

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

$$\frac{\begin{array}{l} T \notin P_D \qquad \Delta'_1 = \cdots = \Delta'_n \\ (P_C; P'_D); \Gamma; \Delta/\Delta'_1; \Omega, y_1{:}X_1 \Uparrow \vdash M_1 : C \qquad y_1 \notin \Delta'_1 \\ \quad \ldots \\ (P_C; P'_D); \Gamma; \Delta/\Delta'_n; \Omega, y_n{:}X_n \Uparrow \vdash M_n : C \qquad y_n \notin \Delta'_n \end{array}}{(P_C; P_D); \Gamma; \Delta/\Delta'_1; \Omega, x{:}T \Uparrow \vdash\ \text{case } x \text{ of } \ldots \mid C_n\ y_n \to M_n : C}\ (\textsc{adt}\mathrm{L})$$

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

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

$$\frac{\textsc{unify}(T_{\overline{\alpha}} \mapsto T_{\overline{\beta}}, \Theta)}{\Theta/\Theta, T_{\overline{\alpha}} \mapsto T_{\overline{\beta}}, P; \Gamma; \Delta/\Delta'; x{:}T_{\overline{\alpha}} \Downarrow \vdash x : T_{\overline{\beta}}}\ (\textsc{adt-init})$$

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

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

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

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

$$\frac{\textsc{getModel}(p) = M}{\Theta/\Theta'; P; \Gamma; \Delta/\Delta' \vdash M : \{a : A \mid p\} \Uparrow}\ (\textsc{ref}\mathrm{R})$$

$$\frac{\textsc{sat}(p_a \Rightarrow p_b)}{\Theta/\Theta'; P; \Gamma; \Delta/\Delta'; x{:}(\{a : A \mid p_a\}) \Downarrow \vdash x : \{b : A \mid p_b\}}\ (\textsc{ref}\mathrm{L})$$

$$\frac{\Theta/\Theta'; P; \Gamma; \Delta; x{:}A \Downarrow \vdash M : C}{\Theta/\Theta'; P; \Gamma; \Delta; x{:}!A \Downarrow \vdash M : !C}\ (!\,{\Downarrow}\mathrm{L})$$

C Examples

Maybe.
Input program:

```
data Maybe a = Nothing | Just a;
data List a = Nil | Cons (a * List a);

synth return :: a -o Maybe a;
synth empty :: Maybe a;
synth bind :: Maybe a -o (a -o Maybe b) -> Maybe b;
synth maybe :: b -> (a -o b) -> Maybe a -o b;
```

Output program:

```
return :: forall a . (a -o Maybe a);
return = Just;

empty :: forall a . Maybe a;
empty = Nothing;

bind :: forall a b . (Maybe a -o (!(a -o Maybe b) -o Maybe b));
bind c d = case c of
    Nothing ->
      let !e = d in Nothing
  | Just f -> let !g = d in g f;

maybe :: forall a b . (!b -o (!(a -o b) -o (Maybe a -o b)));
maybe c d e = let !f = c in
  let !g = d in
    case e of
        Nothing -> f
      | Just h -> g h;
```

List.
Input program:

```
data List a = Nil | Cons (a * List a);
data Maybe a = Nothing | Just a;

synth singleton :: a -o List a;
synth append :: List a -o List a -o List a;
synth map :: (!(a -o b)) -o List a -o List b;
synth foldl :: !(b -o a -o b) -o b -o List a -o b | choose 1;
synth uncons :: List a -o Maybe (a * List a);
synth foldr :: !(a -o b -o b) -o b -o List a -o b;
synth insert :: a -o List a -o List a;
synth concat :: List (List a) -o List a;
```

Ouput program:

```
singleton :: forall a . (a -o List a);
singleton b = Cons (b, Nil);

append :: forall a . (List a -o (List a -o List a));
append b c = case b of
    Nil -> c
  | Cons d ->
      let e*f = d in
        Cons (e, append f c);

map :: forall a b . (!(a -o b) -o (List a -o List b));
map c d = let !e = c in
  case d of
      Nil -> Nil
    | Cons f ->
        let g*h = f in
          Cons (e g, map (!e) h);

foldl :: forall a b . (!(b -o (a -o b)) -o (b -o (List a -o b)));
foldl c d e = let !f = c in
  case e of
      Nil -> d
    | Cons g ->
        let h*i = g in
          foldl (!f) (f d h) i;

uncons :: forall a . (List a -o Maybe (a * List a));
uncons b = case b of
    Nil -> Nothing
  | Cons c -> let d*e = c in Just (d, e);

foldr :: forall a b . (!(a -o (b -o b)) -o (b -o (List a -o b)));
foldr c d e = let !f = c in
  case e of
      Nil -> d
    | Cons g ->
        let h*i = g in
          f h (foldr (!f) d i);

insert :: forall a . (a -o (List a -o List a));
insert b c = case c of
    Nil -> Cons (b, Nil)
  | Cons g ->
      let h*i = g in
```

```
        Cons (h, insert b i);

concat :: forall a . (List (List a) -o List a);
concat b = case b of
    Nil -> Nil
  | Cons c ->
      let d*e = c in
        case d of
            Nil -> concat e
          | Cons k ->
              let l*m = k in
                Cons (l, concat (Cons (m, e)));
```

State. (with a slight optimization that will be added as a control keyword futurely, that allows bind using runState to terminate in a reasonable time)
Input program:

```
data State b a = State (!b -o (a * !b));

synth runState :: State b a -o (!b -o (a * !b));
synth bind :: (State c a -o (a -o State c b) -o State c b) | using (runState);
synth return :: a -o State b a;
synth get :: State a a;
synth put :: !a -o (State a 1);
synth modify :: (!a -o !a) -o State a 1;
synth evalState :: State b a -o !b -o a;
```

Output program:

```
data State b a = State (!b -o (a * !b));

runState :: forall a b . (State b a -o (!b -o (a * !b)));
runState c !f = case c of
    State e ->
        let h*i = e (!f) in
          let !j = i in (h, (!j));

bind :: forall a b c . (State c a -o ((a -o State c b) -o State c b));
bind bs bt = case bs of
    State bu ->
      State (\bv -> let !bw = bv in
                      let cu*cv = bu (!bw) in
                        let !cw = cv in
                          (let dr*ds = runState (bt cu) (!cw) in
                             let !dt = ds in dr, (!cw)));

return :: forall a b . (a -o State b a);
return c = State (\d -> let !e = d in
                (c, (!e)));

get :: forall a . State a a;
get = State (\b -> let !c = b in
                (c, (!c)));

put :: forall a . (!a -o State a 1);
put b = let !c = b in
  State (\d -> let !e = d in
                  ((), (!e)));

modify :: forall a . ((!a -o !a) -o State a 1);
modify b = State (\c -> let !d = c in
                let !f = b (!d) in ((), (!f)));

evalState :: forall a b . (State b a -o (!b -o a));
evalState c d = case c of
    State e ->
      let !f = d in
        let h*i = e (!f) in
          let !j = i in h;
```