Synthesis of Linear Functional Programs

Rodrigo Mesquita and 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 a programmer's productivity and/or program correctness (i.e. through suggestion and/or autocompletion), or as a tool to automate certain parts of the programming process (e.g. in the same way AI might generate some boilerplate text to aid a journalist, rather than to replace the journalist outright), just to name a few examples.

Specifications can take many forms (e.g. polymorphic refinement types [13], examples [6], graded linear types [8]). Regardless of the kind of specification, program synthesizers have to deal with two main inherent sources of complexity: searching over a vast space of (potentially) valid programs, and interpreting user intent. In this work we will explore type-based synthesis where specifications take the form of linear types, which are types that limit resource usage in programs. Concretely, we will explore the proof-theoretic foundations of linear types via the Curry-Howard correspondence with linear logic, viewing program synthesis under the lens of (linear logic) bottom-up proof search.

Type-Based Synthesis. Type-based synthesis uses types as a form of program specification, automatically or semi-automatically producing a program expression with the given type and so matching the specification. In order to make type-based specifications more expressive, type-based synthesis frameworks use rich type systems such as refinement types [13] and polymorphic types [13].

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). However, they can also be a burden – the whole point of these type systems is to ensure that *fewer* programs are deemed well-typed, which can pose additional challenges to the development process. Type-based synthesis leverages rich types as a way of pruning the search space, and by using types the user is given a more "familiar" specification. For instance, the type $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$ 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 $(x{:}\mathsf{Int}) \to (y{:}\mathsf{Int}) \to \{z{:}\mathsf{Int} \mid z = x + y\}$ 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.

This work explores the process of synthesizing linear functional programs from types based on linear logic propositions (i.e. linear types) by leveraging the Curry-Howard correspondence. The correspondence states that propositions in a logic have a direct mapping to types, and well-typed programs correspond to proofs of those propositions. As such, having a type be a proposition in linear logic, we can relate a proof of that proposition directly to a linear functional program — finding a proof is finding a program with that type. Thus, we formulate synthesis as proof search in linear logic, which allows us to inform our work with approaches from the proof search literature. Linear types differ from more traditional types in that they constrain resource usage in programs by *statically* limiting the number of times certain resources can be used during their lifetime. They can be applied to resource-aware programming such as concurrent programming (e.g. session types for message passing concurrency [3]), and to memory-management (e.g. Rust's ownership types).

Goals. In the end, we intend to be able to do full and partial synthesis of well-typed programs. Full synthesis consists of the production of a function (or set of) satisfying the specification; partial synthesis is the "completion" of a partial program (i.e. a function with a *hole* in it). The work will start from a small core linear – i.e. *resource-aware* functional language, building up to recursive types/functions, with potential other avenues of further extension. We will evaluate the work through expressiveness benchmarks (i.e. can we synthesize a term of type T?) and time measurements (i.e. how fast can we synthesize a term of type T?').

2 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 [12] 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 [15]. This relationship, known as the Curry-Howard correspondence (see [16] 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:

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

Elimination rules are easier to think of in a top-down manner. Logically, this rule states that if we prove $A \to B$ 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 $A \to B$ 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 [7] 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 into 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.

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 [4,10], 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.

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.

Focusing. Even with everything mentioned so far, non-determinism is still present proof-search in logic, e,g., at any given point, many proof rules are applicable in general. The technique of focusing [1,5] 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 search is necessary.

3 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 [8,13,11,6] 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 respective corresponding logics and programming 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 [14]). Some other common patterns we might want to follow are the use of type refinements [13] and the support for synthesis of recursive functions [13,11]. These works share some common ground, and base themselves of the same works we do – most famously the Curry-Howard correspondence, but also, for example, works on focusing-based proof-search [1].

Type-and-Example-Directed Program Synthesis. This work [11,6] explores a purely functional, recursive program synthesis technique based on types/proof search (as we intend to do), with examples as an auxiliar technique to trim down the program synthesis vast search space. A good use is made of the general strategy of using a type system to generate programs (by "inverting" the type system), rather than using it to type-check one. Moreover, the paper uses a data structure called "refinement tree" in conjunction with the extra information provided by the examples to allow for efficient synthesis of non-trivial programs. Overall it employs good engineering to achieve type-based synthesis of functional typed programs, with at least two additions that we might want to follow too (type refinements and recursive functions).

Program Synthesis from Polymorphic Refinement Types. The work [13] also studies synthesis recursive functional programs but in a more "advanced"

setting. The specification is a combination of two rich forms of types: polymorphic and refinement types (which correspond to a first-order logic through the Curry-Howard isomorphism) – this is an interesting combination that offers "expressive power and decidability, which enables automatic verification–and hence synthesis–of nontrivial programs". Their approach to refinement types consists a new algorithm that supports decomposition of the refinement specification, allowing for separation between the language of specification and programs and making the approach amenable to compositional synthesis techniques.

Resourceful Program Synthesis from Graded Linear Types. The work [8] synthesizes programs using an approach very similar to our own. The work employs so-called graded modal types, which is an refinement of pure linear types – a more *fine-grained* version, since it 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 resource management approach [4,9] – the model we will be using.

They also use focusing as a solution to trim down search space and to ensure that synthesis only produces well-typed programs, and our solution will use the same literature as a base. However, since their underlying logic is *modal* rather than purely *linear*, it lacks a clear correspondence with concurrent session-typed programs [3,2], which is a crucial avenue of future work. Moreover, their use grading effectively requires constraint solving to be integrated with the synthesis procedure, which can limit the effectiveness of the overall approach as one scales to more sophisticated settings (e.g. refinement types).

4 Goals and Work Plan

We start with the definition of our type system and simply-typed functional language with linear function, sum and product types ($\multimap$, $\otimes$, $\mathbf{1}$, $\&$, $\oplus$, $!$). Through the Curry-Howard correspondence, we'll formulate the synthesis of typed programs in our functional language as bottom-up proof-search in linear logic.

Since proof rules in *natural deduction* do not entail a proof-search strategy, we will use the equivalent *sequent calculus* system to formulate our typing rules in a way that is amenable to proof search. In sequent calculus, rules can be naturally understood in a *bottom-up* manner, simplifying the reasoning for the synthesizer. Moreover, the linear nature of our system requires a way to algorithmically reason about resource contexts: for instance, if we read rule $\otimes I$ from Section 2 bottom-up, we must non-deterministically guess how to correctly split ambient resources into Δ_1 and Δ_2 to separately prove A and B. The resource-management approach we'll adopt to solve this has been previously explained in more detail.

With the use of input/output contexts, and the sequent calculus formulation of the inference rules, we draw closer to an effective strategy for bottom-up

proof search and so of program synthesis. However, the search process still encompasses significant non-determinism. To address this we draw on the technique of focusing [1,5] as discussed above.

With the basic framework in place, we will consider additional (simply-typed) language constructs and synthesis of program fragments or *partial synthesis* – given a program with a typed hole, synthesize a term to fill out the hole accordingly. We will subsequently also consider synthesis of recursive functions and, might, later on, add refinement types also as a way to constrain the valid programs space.

Putting together everything above, and the more detailed information in the background section – linear types, sequent calculus, resource management, and focusing, we have a set of inference rules (written out in the appendix) that define how the synthesis process will progress.

This is the theory we're interested in and have studied so far, which will allow us to build the actual synthetizer. From here, our work plan gives an overview of the steps we'll take.

4.1 Work Plan

The end goal is to have a synthetizer capable of taking a type (possibly refined) as input, and to produce (possibly interactively) one or more functions with that type, if any exist.

As mentioned above, we will begin with (both partial and full) synthesis for the linear λ-calculus. We will then extend the language with additional type constructs (e.g. base types, records, algebraic data types) and consider synthesis in that setting. Finally, we will consider synthesis of recursive functions, and, time-permitting, we will consider simple forms of type refinements (e.g. arithmetic refinements).

We've already implemented a parser and a type-checker for a linear λ-calculus using the resource management technique described in the previous section – more than good preparation for the development of the synthesizer, they will be useful to *validate* the synthesized functions, and to enable the synthesis of partial programs. The implementations have been carried out in Haskell since it's easier to model and implement inference rules in a functional setting.

Step by step. Our work so far has been constantly accompanied by the study of somewhat dense theoretical material, state of the art, which we must understand to overcome challenges in these fields, in order to develop a more scalable synthetizer – we want to display concrete results to prove the feasability of program synthesis from linear types, and thus open the possibility for future application to related fields, such as program synthesis based on session types for message-passing concurrency.

Apart from the theory, when we started studying type systems, we created a simple type-checker for the simply-typed λ-calculus. While learning about linear type systems, and the simply-typed linear λ-calculus, we developed a parser, a desugaring module, and an evaluation module to write, analyze and

execute simple programs in the language; culminating in writing a type-checker for the linear λ-calculus. At this point, we can parse, typecheck, and execute this small linear functional language. The next step is to start developing the actual synthetizer, embedding it with all the techniques we've seen. Following work will be done to augment usability, expand the language domain, and add features such as those mentioned above.

Evaluation. Validation and evaluation of our work is fairly easy. Simple questions such as *does it typecheck*? *can we synthesize*? *how fast*? are the core of our evaluation. In more concrete terms, we can compare our times to state of the art synthetizers, present a list of working examples of programs synthesized by our framework, and corresponding benchmarks.

On the eventual addition of recursive functions, refinement types, and interactive synthesis, we'll consider, respectively, other examples to showcase recursive functions synthesis, updated times for benchmarks with refinement types, and a display of the possible user interaction in the user process, and the resulting function following it.

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. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR 2010. pp. 222–236 (2010)
3. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
4. 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`
5. 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`
6. 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`
7. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
8. 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
9. 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
10. 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
11. 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
12. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
13. 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
14. 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
15. Prawitz, D.: Natural Deduction. Almquist & Wiksell (1965)
16. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). https://doi.org/10.1145/2699407, https://doi.org/10.1145/2699407