Programming Assistance for Type-Directed Programming (Extended Abstract)

Peter-Michael Osera
Grinnell College, USA
osera@cs.grinnell.edu

Abstract

Type-directed programming is a powerful programming paradigm where rich types dictate the structure of the program, making design largely automatic. While mechanical, this paradigm still requires manual reasoning that is both tedious and error-prone. We propose using type-directed program synthesis techniques to build an interactive programming assistant for type-directed programming. This tool bridges the gaps between simple auto-completion engines and program synthesis, complementing the strengths of each.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (Functional) Languages; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Program Editors; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis

Keywords Type-directed Programming, Program Synthesis

1. The Spectrum of Assistance to Synthesis

"My program just writes itself!" This phrase is frequently uttered by programmers as they come to realize the power of typed functional programming languages. In such languages, their type systems are strong enough to greatly constrain the set of possible programs that the developer can write. For example, consider writing a program that has type `int -> string`. If we have two functions, `f : int -> bool` and `g : bool -> string`, then by inspecting the types, we may consider the program `let f (x:int) : string = g (f x)`, the composition of `f` and `g`. This is, of course, not the only possible function; there are an infinite number of them. However, consider writing a program that has the richer polymorphic type `('a -> 'b) -> 'a list -> 'b list`. There are far fewer programs that one can write of this type, only one of which (the canonical `map` function) that does something reasonable with its inputs.

While a developer may feel like writing a richly-typed program is automatic, the actual process is anything but. The developer must reason carefully about the programming constructs, operations, and functions available to her and deduce how to put them all together into a final program. In a language with many available components, she may find it difficult and tedious to keep track of all them and comb through the various possibilities. Ultimately, this process is mechanical. Therefore, it ought to be possible for our programming environment—languages, editors, and tools—to support reasoning about program design in a type-directed fashion.

Two classes of approaches have been proposed in the past to assist developers in type-directed program development. Type-based auto-completion tools (Perelman et al. 2012; Gvero et al. 2013) use types to identify candidate components—usually function calls—that could complete a given program fragment. Goal-refinement programming tools such as the typed holes feature of Haskell (inspired by similar features found in automated theorem provers like Coq and Agda) give the user information about the context—the set of available components and their types—at the current point in the program. However, these two approaches fall short in distinct dimensions. Goal-refinement tools help the developer manage the complexity of types but do not automate the reasoning she must perform to write the final program. Type-based auto-completion tools typically have limited scope; they can only analyze function calls or similar expressions and cannot handle more complex programming constructs such as conditionals. Ideally, we would like a tool that can handle the full breadth of our programming language while automating the reasoning process in some way.

Fully automating the type-directed programming process is a form of *program synthesis*. Program synthesis is the automatic generation of programs from specification. Recently, researchers have investigated the use of type theory as a means to optimize the synthesis process (Feser et al. 2015; Osera and Zdancewic 2015; Scherer and Rèmy 2015; Frankle et al. 2016; Polikarpova et al. 2016). These techniques all take advantage of the constraining nature of rich types to narrow the search space of possible programs to a manageable size of candidates that can then be filtered through other specification, *e.g.*, input-output examples denoting how the desired program should behave.

While researchers have made substantial advances in program synthesis over the last thirty years, the problem still proves to be a difficult one to completely solve. Because the set of possible programs grows exponentially with program size and there are (usually) an infinite number of such candidates to explore, general-purpose synthesizer, type-directed or otherwise, can only synthesize programs of modest size. To get around this problem, researchers typically target a narrower domain of programs, sacrificing expressiveness for tractability. This approach has seen great success but does not solve the synthesis problem for general-purpose programming.

Rather than narrowing the scope of the synthesizer, we propose a *semi-automated synthesis approach* for type-directed programming, bridging the gap between the previous work in type-directed programming assistance and general program synthesis. Our current prototype, SCOUT, uses the MYTH synthesizer (Osera and Zdancewic 2015) to power an `emacs` plugin that assists a programmer in developing OCaml functions. It does this by displaying possible program skeletons based on a specification of types and ex-

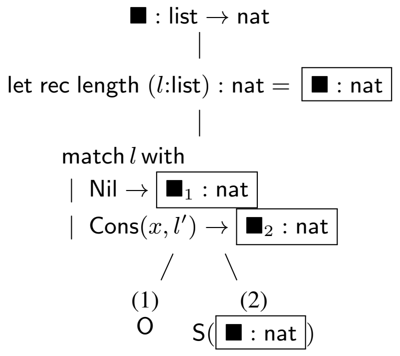
[Copyright notice will appear here once 'preprint' option is removed.]

amples. The programmer interacts with the tool by specifying which of these skeletons she would like to explore further, substantially pruning the search space of programs the synthesizer must consider.

2. SCOUT: Interactive Refinement with Types

The core of the MYTH synthesizer utilizes a type-theoretic, foundational approach to program synthesis where the "inputs" and "outputs" of the typing relation of a language, written $\Gamma \vdash e : \tau$, are swapped and augmented with example refinement to create a program synthesis relation $\Gamma \vdash \tau \triangleright X \rightsquigarrow e$. To turn this relation into a synthesis algorithm, MYTH employs a data structure called a *refinement tree* which encodes the shape of potential programs that could be synthesized according to the types and examples that the user provides. These shapes are drawn from the *introduction forms* of the language, the expressions that introduce values of a given type, *e.g.*, functions. They are determined entirely by the synthesis specification and thus can be discovered quickly by the synthesizer. In contrast, *elimination forms* are expressions that consume values of a given type, *e.g.*, function application. MYTH enumerates elimination forms in a separate, more time-consuming pass in order to complete the refinement tree and create a final program.

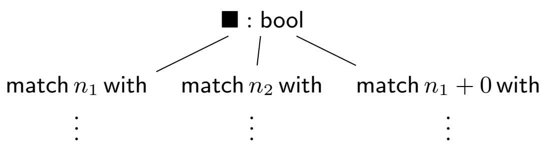
Here is an example refinement tree that might be created while synthesizing a list length function:



■ : list → nat
let rec length (l:list) : nat = ■ : nat
match l with
\| Nil → ■₁ : nat
\| Cons(x, l′) → ■₂ : nat
(1)
O
(2)
S(■ : nat)

Nodes in the tree correspond to *synthesis goals*: partial programs with typed holes written $\blacksquare : \tau$. For example, the root denotes the initial goal of synthesizing a program of type list $\rightarrow$ nat. Its child denotes solving this goal by synthesizing an introduction form, here a function, leaving the body of the function as a synthesis subproblem that must be solved. The synthesizer incrementally grows the tree by exploring the set of possible introduction forms of the hole's type, refining the types and examples in the process. Holes left unfilled, *e.g.*, the "$\blacksquare$: nat" goal in the Cons branch of the match, are filled in by the synthesizer in a separate pass where raw elimination forms are enumerated (in order of increasing program size) and checked against the examples.

For complex programs, the refinement tree may possess many branches with many sub-synthesis problems to solve, and the leaves of the tree may require large elimination forms to be guessed that take a significant amount of time to enumerate. This leads to the exponential performance bottleneck described in the previous section. For example, a node in the refinement tree may be fulfilled by many match expressions



some of which are plausible but others are redundant or unfruitful.

Because MYTH is fully automatic, it is forced to explore all of these possibilities. However, using SCOUT, the user provides additional insight to the synthesizer by choosing which of these possible refinements to explore, implicitly pruning out the other branches in the process. With the example above, SCOUT reports the types of the variables in the context and the refined examples at this point in the program. The user then uses this information to choose one of the three possible refinements to explore, *e.g.*, the match that analyzes n_1. When the synthesizer now tries to fill in the remaining holes, it only needs to fill in the program skeleton user chose, a dramatic reduction of the search space.

3. Current Status and Future Directions

We are currently developing the initial SCOUT prototype to investigate the effectiveness of this semi-automated type-directed approach to program development. As a program synthesis tool, we hypothesize that user-driven pruning of the refinement tree will allow SCOUT to synthesize much larger programs than MYTH could handle, and as a program assistance tool, we hypothesize that the information and automation that the tool provides will help functional programmers write code in a type-directed style much more efficiently. Our initial experimentation with our prototype has shown great promise with respect to both these hypotheses.

Finally, while developing SCOUT, we have begun to ask ourselves a number of additional questions about the usability of type-directed programming tools and synthesizers. In particular: 1. What is an appropriate balance between ease-of-use, speed, and automation for a programming assistant? 2. What information does the user need to productively guide an interactive synthesizer? In addition to usability, the interactive nature of SCOUT has also led to some interesting questions about our synthesis techniques. Because the tool is now operating in an interactive setting, the synthesizer must behave gracefully when the user edits the code outside of the synthesizer. Simply re-parsing the world and starting from scratch is not acceptable because synthesis is a time-consuming process. We instead need to preserve as much synthesis information as possible when updating the synthesizer's state in light of these edits.

References

J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, POPL 2016, 2016.

T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

G. Scherer and D. Rèmy. Which simple types have a unique inhabitant? In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2015.