TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Tavo Annus 211564IAPM

Term Search in Rust

Master's thesis

Supervisor

Philipp Joram
MSc

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Tavo Annus 211564IAPM

Avaldise otsing programmeerimiskeeles Rust

Magistritöö

Juhendaja

Philipp Joram
MSc

Tallinn 2024

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Tavo Annus

Date: 12.05.2024

Abstract

Rust is a general-purpose systems programming language that guarantees memory safety without the need for a garbage collector. With on-par performance with C/C++, Rust attempts to challenge C/C++'s position in the market by providing better tooling and a better developer experience.

The Rust type system is similar to many functional programming languages as it features a rich type system, including sum and product types. Developer experience is more similar to that of high-level functional programming languages than C/C++. However, Rust does not have a tool for term search – automatic program synthesis based on types. Yet we believe it is a perfect candidate for one with its expressive type system.

In this thesis, we extend the official Rust language server `rust-analyzer` with a term search tool. In addition to program synthesis, we experiment with using term search for autocompletion. We develop the algorithm for term search in three iterations. The first iteration is the simplest algorithm that follows one that is used in Agsy, a similar tool for Agda. The second iteration improves on the first by reversing the search direction, simplifying the caching of intermediate results. In the final iteration, we implement a bidirectional search. This algorithm can synthesize terms in many more situations than in previous iterations, without a significant decrease in performance.

To evaluate the performance of our algorithm, we run it on 155 popular open-source Rust libraries. We delete parts of their source code, creating "holes". We then let the algorithm resynthesize the missing code, and measure how many holes the algorithm can fill and how often the algorithm suggests the original code.

As an outcome of this thesis, we also upstream our tool to the official `rust-analyzer`.

The thesis is written in English and is 51 pages long, including 6 chapters, 7 figures 28 code listings and 4 tables.

Annotatsioon

Rust on üldotstarbeline programmeerimiskeel nii süsteemi kui ka rakenduste tarkvara loomiseks. Erinevalt teistest programmeerimiskeeltest suudab Rust garanteerida turvalise mälu kasutuse kasutamata spetsiaalset mäluhaldurit. Rust konkureerib programmeerimiskeeltega C ja C++ pakkudes arendajale paremaid tööriistu ja paremat arenduse kogemust järele andmata programmi kiiruses.

Rusti tüübisüsteem sarnaneb funktsionaalsete keelte tüübisüsteemidele sisaldades algebralisi andmetüüpe. Arendaja kogemus Rustis sarnaneb seetõttu rohkem kogemusele kõrgematasemelistes funktsionaalsetes programmeerimiskeeltes kui kogemusele C/C++ arenduses. Samas Rustis puudub hetkel tööriist, mis suudaks automaatselt genereerida avaldisi, mis vastavad oodatud tüübile. Sellised tööriistad on tavalised funktsionaalsetes programmeerimiskeeltes ja me usume, et ka Rustil on taolisest tööriistast kasu.

Käesolevas töös arendame edasi Rusti ametlikku tööriista `rust-analyzer`, lisades sellele võimekuse genereerida avaldisi tüüpide alusel. Lisaks programmide genereerimisele uurime, kas avaldise otsingut on võimalik kasutada ka targema automaatse sõnalõpetuse loomiseks. Me arendame oma algoritmi kolme iteratsioonina. Esimene iteratsioon on kõige lihtsam ja sarnaneb suuresti algoritmile mida kasutatakse Agsys, sarnases tööriistas Agda jaoks. Teises iteratsioonis arendame algoritmi edasi muutes otsingu suuna vastupidiseks. Sooritades otsingut vastupidises suunas saame lihtsalt lisada vahepealsete väärtuste puhverdamise ning teisi optimeerimisi. Viimases versioonis muudame otsingu kahesuunaliseks. Sooritades otsingut kahes suunas, suudame leida avaldise rohkemates kohtades ilma algoritmi tööd oluliselt aeglustamata.

Töö tulemuste hindamiseks jooksutame me algoritmi 155-l vabavaralisel Rusti programmil. Kustutame osa olemasolevast lähtekoodist, jättes programmi koodi "augud". Nüüd kasutame oma algoritmi, et genereerida kood nende aukude jaoks. Mõõdame kui palju auke suudab algoritm täita ja kui tihti suudab algoritm genereerida tagasi originaalse lähtekoodi.

Töö väljundina saadame oma algoritmi ametlikku `rust-analyzer`'i, et seda saaksid kasutada kõik Rusti arendajad.

Lõputöö on kirjutatud inglise keeles keeles ning sisaldab teksti 51 leheküljel, 6 peatükki, 7 joonist 28 koodinäidist ja 4 tabelit.

Contents

1 Introduction

Rust[1] is a programming language for developing reliable and efficient systems. The language was created by Graydon Hoare, later developed at Mozilla for Firefox, but is now gaining popularity and has found its way to the Linux kernel[2]. It differs from other popular systems programming languages such as C and C++ by focusing more on the reliability and productivity of the programmer. Rust has an expressive type system that guarantees a lack of undefined behavior at compile type. It is done with a novel ownership model and is enforced by a compiler tool called borrow checker. The borrow checker rejects all programs that may contain illegal memory accesses or data races.

We will call the set of programs that can be compiled valid, as they are guaranteed to not cause undefined behavior. Many programming languages with type systems that guarantee the program to be valid have tools that help the programmer with term search i.e. by searching for valid programs (usually called expressions in Rust) that satisfy the type system. Rust, however, does not have tools for term search, although the type system makes it a perfect candidate for one. Consider the following Rust program in Listing 1:

```rust
enum Option<T> { None, Some(T) }

fn wrap(arg: i32) -> Option<i32> {
    todo!();
}
```

Listing 1: Rust function to wrap `i32` in `Option`

From the types of values in scope and constructors of `Option`, we can produce the expected result for `todo!()` by applying the constructor `Some` to `arg` and returning it. By combining multiple constructors for types as well as functions in scope or methods on types, it is possible to produce more complex, valid programs.

1.1 Motivation

Due to Rust's expressive type system, programmers might find themselves quite often wrapping the result of some function behind multiple layers of constructors. For example, in the web backend framework `actix-web`[3], a typical JSON endpoint function might look something like shown in Listing 2.

```rust
struct FooResponse { /* Struct fields */ }

#[get("/foo")]
async fn foo() -> Option<Json<FooResponse>> {
    let service_res = service.foo(); // Get a response from some service
    Some(Json(FooResponse {
        /* Fill struct fields from `service_res` */
    }))
}
```

Listing 2: Example endpoint in `actix-web` framework

We can see that converting the result `service_res` from the service to `FooResponse` and wrapping it in `Some(Json(...))` can be automatically generated just by making the types match.

[1] Rust, Accessed: 2024-04-06, URL: https://www.rust-lang.org/
[2] Linux 6.1-rc1, Accessed: 2024-04-06, URL: https://lkml.org/lkml/2022/10/16/359
[3] Actix, Accessed: 2024-04-06, URL: https://actix.rs/

This means that term search can be used to reduce the amount of code the programmer has to write.

When investigating common usage patterns among programmers using large language models for code generation, [1, p. 19] found two patterns:

1. Language models are used to reduce the amount of code the programmer has to write therefore making them faster. They call it the *acceleration mode*.
2. Language models are used to explore possible ways to complete incomplete programs. This is commonly used when a programmer is using new libraries and is unsure how to continue. They call this usage pattern *exploration mode*.

We argue that the same patterns can be found among programmers using term search.

In acceleration mode, term search is not as powerful as language models, but it can be more predictable as it has well-defined tactics that it uses rather than deep neural networks. There is not so much "wow" effect - it just produces code that one could write by trying different programs that type-check.

However, we expect term search to perform well in *exploration mode*. [1, p. 10] finds that programmers tend to explore only if they have confidence in the tool. As term search only produces valid programs based on well-defined tactics, it is a lot easier to trust it than code generation based on language models that have some uncertainty in them.

1.2 Research Objectives

The main objective of this thesis is to implement a tactics-based term search for the programming language Rust. The algorithm should:

- only produce valid programs, i.e. programs that compile
- finish fast enough to be used interactively while typing
- produce suggestions for a wide variety of Rust programs
- not crash or cause other issues on any Rust program

Other objectives include:

- Evaluating the fitness of tactics on existing large codebases
- Investigating term search usability for auto-completion

1.3 Contributions of the thesis

In this thesis, we make the following contributions:

- Section 2 gives an overview of term search algorithms used in other languages and autocompletion tools used in Rust and mainstream programming languages. We also introduce some aspects of the Rust programming language that are relevant to the term search.
- Section 3 introduces term search to Rust by extending the official language server of the Rust programming language, `rust-analyzer`. We discuss the implementation of the algorithm in detail as well as different use cases. In Section 3.4, we describe the capabilities of our tool.
- Section 4 evaluates the performance of the tool. We compare it to mainstream tools, some machine-learning-based methods and term search tools in other programming languages.
- Section 5 describes future work that would improve our implementation. This includes technical challenges but also describes possible extensions to the algorithm.

We have upstreamed our implementation of term search to the `rust-analyzer` project. It is part of the official distribution since version `v0.3.1850`[1], released on February 19th 2024. An archived version can be found at the Software Heritage Archive `swh:1:rev:6b250a22c41b2899b0735c5bc607e50c3d774d74` .

[1]Rust Analyzer Changelog #221, Accessed: 2024-04-06, URL: https://rust-analyzer.github.io/thisweek/2024/02/19/changelog-221.html

2 Background

In this chapter, we will take a look at the type system of the Rust programming language to understand the context of our task. Next, we will take a look at what the term search is and how it is commonly used. Later, we will study some implementations for term search to better understand how the algorithms for it work. In the end, we will briefly cover how *autocompletion* is implemented in modern tools to give some context of the framework we are working in and the tools that we are improving on.

2.1 The Rust language

Rust is a general-purpose systems programming language first released in 2015[1]. It takes lots of inspiration from functional programming languages, namely, it supports algebraic data types, higher-order functions, and immutability.

2.1.1 Type system

Rust has multiple different kinds of types. There are scalar types, references, compound data types, algebraic data types, function types, and more. In this section, we will discuss types that are relevant to the term search implementation we are building. We will ignore some of the more complex data types such as function types as implementing term search for them is out of the scope of this thesis.

Scalar types are the simplest data types in Rust. A scalar type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, booleans, and characters.

Compound types can group multiple values into one type. Rust has two primitive compound types: arrays and tuples. An array is a type that can store a fixed amount of elements of the same type. Tuple, however, is a type that groups values of different types. Examples for both array and tuple types can be seen in Listing 3 on lines 2 and 3.

Reference types are types that contain no other data than a reference to some other type. An example of a reference type can be seen in Listing 3 on line 4.

```
1 let a: i32 = 0; // Scalar type for 32 bit signed integer
2 let b: [u8, 3] = [1, 2, 3]; // Array type that stores 3 values of type `u8`
3 let c: (bool, char, f32) = (true, 'z', 0.0); // Tuple that consists of 3 types
4 let d: &i32 = &a; // Reference type to `i32`
```

Listing 3: Types in Rust

Rust has two kinds of algebraic types: *structures* (also referred as `structs`) and *enumerations* (also referred as `enums`). Structures are product types, and enumerations are sum types. Each of them comes with their own data constructors. Structures have one constructor that takes arguments for all of its fields. Enumerations have one constructor for each of their variants.

Both of them are shown in Listing 4.

[1] Announcing Rust 1.0, Accessed: 2024-04-06, URL: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html

```
1  // Product type has values for both `x` and `y`
2  struct Foo {
3    x: i32,
4    y: bool,
5  }
6
7  // Sum type has values for either constructor `A` or `B`
8  enum Bar {
9    A(i32),
10   B(bool),
11 }
12
13 fn main() {
14   let foo = Foo { x: 1, y: true };  // Initialize struct
15   let bar = Bar::B(false);          // Initialize enum with one of it's variants
16 }
```

Listing 4: Sum and product types in Rust

To initialize a `struct`, we have to provide terms for each of the fields it has, as shown on line 12. For `enum`, we choose one of the variants we wish to construct and only need to provide terms for that variant. Note that structures and enumeration types may both depend on generic types, i.e. types that are specified at the call site rather than being hard-coded to the type signature. For example, in Listing 5, we made the struct `Foo` generic over `T` by making the field `x` be of generic type `T` rather than some concrete type. A common generic enum in Rust is the `Option` type which is used to represent optional values. The `None` constructor takes no arguments and indicates that there is no value. Constructor `Some(T)` takes one term of type `T` and indicates that there is some value stored in `Option`. Initializing structs and enums with different types is shown in the `main` function at the end of Listing 5.

```
1  struct Foo<T> {
2    x: T,
3    y: bool,
4  }
5
6  enum Option<T> {
7    Some(T),
8    None,
9  }
10
11 fn main() {
12   let foo_bool: Foo<bool> = Foo { x: true, y: true};
13   let foo_int: Foo<i32> = Foo { x: 123, y: true};
14   let option_str: Option<&str> = Some("some string");
15   let option_bool: Option<bool> = Some(false);
16 }
```

Listing 5: Sum and product types with generics

2.1.2 Type unification

It is possible to check for either syntactic or semantic equality between the two types. Two types are syntactically equal if they have the same syntax. Syntactic equality is a very restrictive way to compare types. A much more permissive way to compare types is semantic equality. Semantic equality of types means that two types contain the same information and can be used interchangeably.

Using syntactic equality to compare types can cause problems. Rust high-level intermediate representation (HIR) has multiple ways to define a type. This means that the same type can be defined in multiple ways that are not syntactically equal.

For example, in the program `type Foo = i32`, the type `Foo` and the type `i32` are not syntactically equal. However, they are semantically equal, as `Foo` is an alias for `i32`. This means that the types unify even though they are syntactically different.

To check for semantic equality of types we see if two types can be unified. Rust's type system is based on a Hindley-Milner type system [2], therefore the types are compared in a typing environment. In Rust, the *trait solver* is responsible for checking the unification of types[1]. The trait solver works at the HIR level of abstraction, and it is heavily inspired by Prolog engines. The trait solver uses "first-order hereditary harrop" (FOHH) clauses, which are Horn clauses that are allowed to have quantifiers in the body [3].

Unification of types `X` and `Y` is done by registering a new clause `Unify(X, Y)` (the *goal*) and solving for it. Solving is done by a Prolog-like engine, which tries to satisfy all clauses registered in the typing environment. If a contradiction is found between the goal and the clauses, there is no solution, and the types `X` and `Y` do not unify. If a solution is found, it contains a set of subgoals that still need to be proven. If we manage to recursively prove all the subgoals, then we know that `X` and `Y` unify. If some goals remain unsolved but there is also no contradiction, then simply more information is needed to guarantee unification. How we treat the last case depends on the use case, but in this thesis, for simplicity, we assume that the types do not unify. An example of unification can be seen in Listing 6.

```
1  trait HasAnswer {
2    type Answer;
3    fn get_answer(&self) -> Self::Answer;
4  }
5
6  struct Life { /* fields */ }
7  impl HasAnswer for Life {
8    type Answer = u8;
9    fn get_answer(&self) -> Self::Answer { 42 }
10 }
11
12 fn main() {
13   let life = Life { /* fields */ };
14   let ans: u8 = life.get_answer();
15   assert!(ans == 42);
16 }
```

Listing 6: A unification problem for the return type of `life.get_answer()`. The goal is `Unify(<Life as HasAnswer>::Answer, u8)`. In context is `Implemented(Life: HasAnswer)` and `AliasEq(<Life as HasAnswer>::Answer = u8)`. From these clauses, we can solve the problem.

2.1.3 Borrow checking

Another crucial step for the Rust compiler is borrow checking[2]. The main responsibilities of the borrow checker are to make sure that:

- All variables are initialized before being used

[1] Rust Compiler Development Guide, The ty module: representing types, Accessed: 2024-04-06, URL: https://rustc-dev-guide.rust-lang.org/ty.html

[2] Rust Compiler Development Guide, MIR borrow check, Accessed: 2024-04-06, URL: https://rustc-dev-guide.rust-lang.org/borrow_check.html

- No value is moved twice or used after being dropped
- No value is moved while borrowed
- No immutable variable can be mutated
- There can be only one mutable borrow

Some examples of kinds of bugs that the borrow checker prevents are *use-after-free* and *double-free*

The borrow checker works at the Middle Intermediate Representation (MIR) level of abstraction. The currently used model for borrows is Non-Lexical Lifetimes (NLL). The borrow checker first builds up a control flow graph to find all possible data accesses and moves. Then it builds up constraints between lifetimes. After that, regions for every lifetime are built up. A region for a lifetime is a set of program points at which the region is valid. The regions are built up from constraints:

- A liveness constraint arises when some variable whose type includes a region R is live at some point P. This simply means that the region R must include point P.
- Outlives constraint `'a: 'b` means that the region of `'a` has to also be a superset of the region of `'b`.

From the regions, the borrow checker can calculate all the borrows at every program point. An extra pass is made over all the variables, and errors are reported whenever aliasing rules are violated.

Rust also has a concept of two-phased borrows that splits the borrow into two phases: reservation and activation. These are used to allow nested function calls like `vec.push(vec.len())`. These programs would otherwise be invalid, as in the example above `vec.len()` is immutably borrowed while `vec.push(...)` takes the mutable borrow. The two-stage borrows are treated as follows:

- It is checked that no mutable borrow conflicts with the two-phase borrow at the reservation point (`vec.len()` for the example above).
- Between the reservation and the activation point, the two-phase borrow acts as a shared borrow.
- After the activation point, the two-phase borrow acts as a mutable borrow.

There is also an option to escape the restrictions of the borrow checker by using `unsafe` code blocks. In an `unsafe` code block, the programmer has the sole responsibility to guarantee the validity of aliasing rules with no help from the borrow checker.

2.2 Term search

Term search is the process of generating terms that satisfy some type in a given context. In automated theorem proving, this is usually known as proof search. In Rust, we call it a term search, as we don't usually think of programs as proofs.

The Curry-Howard correspondence is a direct correspondence between computer programs and mathematical proofs. The correspondence is used in proof assistants such as Coq and Isabelle and also in dependently typed languages such as Agda and Idris. The idea is to state a proposition as a type and then prove it by producing a value of the given type, as explained in [4].

For example, if we have an addition on natural numbers defined in Idris as shown in Listing 7.

```
add : Nat -> Nat -> Nat
add Z     m = m
add (S k) m = S (add k m)
```

Listing 7: Addition of natural numbers in Idris

We can prove that adding any natural number `m` to 0 is equal to the natural number `m`. For that, we create a declaration `add_zero` with the type of the proposition and prove it by defining a program that satisfies the type.

```
add_zero : (m : Nat) -> add Z m = m  -- Proposition
add_zero m = Refl                    -- Proof
```

Listing 8: Prove $0 + n = n$ in Idris

The example above is quite trivial, as the compiler can figure out from the definition of `add` that `add Z m` is defined to be `m` according to the first clause in the definition of `add` Based on that we can prove `add_zero` by reflexivity. However, if there are more steps required, writing proofs manually gets cumbersome, so we use tools to automatically search for terms that inhabit a type i.e. proposition. For example, Agda has a tool called Agsy that is used for term search, and Idris has this built into its compiler.

2.2.1 Term search in Agda

Agda [5] is a dependently typed functional programming language and proof assistant. It is one of the first languages that has sufficiently good tools for leveraging term search for inductive proofs. We will be more interested in the proof assistant part of Agda, as it is the one leveraging the term search to help the programmer come up with proofs. As there are multiple options, we picked two that seem the most popular or relevant for our use case. We chose Agsy as this is the well-known tool that is part of the Agda project itself, and Mimer, which attempts to improve on Agsy.

Agsy

Agsy is the official term-search-based proof assistant for Agda. It was first published in 2006 in [6] and integrated into Agda in 2009[1].

We will be looking at the high-level implementation of its algorithm for term search. In principle, Agsy iteratively refines problems into more subproblems until enough subproblems can be solved. This process is called iterative deepening. This is necessary as a problem may, in general, be refined to infinite depth.

The refinement of a problem can produce multiple branches with subproblems. In some cases, we need to solve all the subproblems. In other cases, it is sufficient to solve just one of the subproblems to solve the "top-level" problem. An example where we need to solve just one of the subproblems is when we try different approaches to come up with a term. For example, we can either use some local variable, function or constructor to solve the problem as shown in Figure 1.

[1] Agda, Automatic Proof Search (Auto), Accessed: 2024-04-06, URL: https://agda.readthedocs.io/en/v2.6.4.1/tools/auto.html

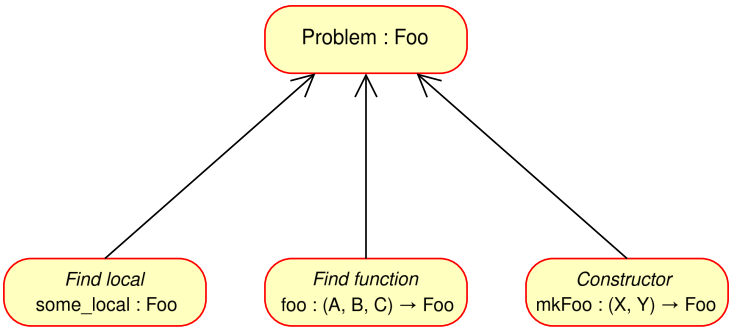

Figure 1: Solving the top-level problem requires solving *at least one* of the subproblems

In case we use constructors or functions that take multiple arguments, we need to solve all the subproblems of finding terms for arguments. The same is true for case splitting: we have to solve subproblems for all the cases. For example, shown in Figure 2 we see that function `foo : (A, B, C) -> Foo` can only be used if we manage to solve the subproblems of finding terms of the correct type for all the arguments.

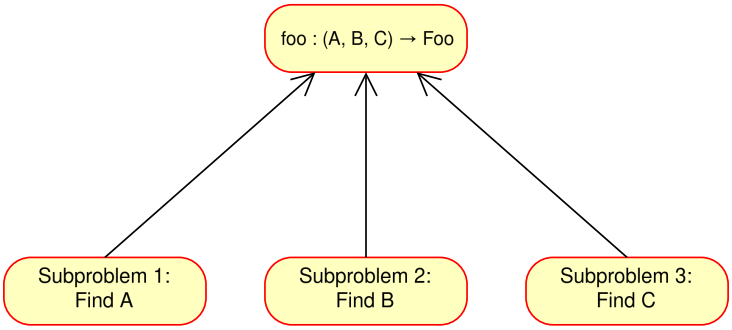

Figure 2: Solving the top-level problem requires solving *all* of the subproblems

Agsy uses problem collections (`PrbColl`) to model the subproblems that need to be all solved individually for the "top-level" problem to be solved. Solution collections (`SolColl`) are used to keep track of solutions for a particular problem collection. A solution collection has a solution for each of the problems in a corresponding problem collection.

The intuition for the tool is the following:

1. Given a problem, we create a set of possible subproblem collections out of which we need to solve *at least one*, as shown in Figure 1.
2. We attempt to solve *all* the subproblem collections by recursively solving all the problems in the collection
3. If we manage to solve *all* the problems in the collection, we use it as a possible solution, otherwise, we discard it as a dead end.

The algorithm itself is based on depth-first search (DFS) and consists of two subfunctions. Function `search: Problem -> Maybe [Solution]` is the main entry point that attempts to find a set of solutions for a problem. The function internally makes use of another function `searchColl: PrbColl -> Maybe [SolColl]` that attempts to find a set of solution collections for a problem collection. The pseudocode for the `search` and `searchColl` functions can be seen in Listing 9.

We model Problem collections as a list of subproblems together with a *refinement* that produces those problems. A refinement is a recipe to transform the problem into zero or more subprob-

lems. For example, finding a pair `(Bool, Int)` can be refined to two subproblems: finding a term of type `Bool` and another of type `Int` and applying the tuple constructor `(_,_)`. If we refine the problem without creating any new subproblems, then we can call the problem solved. Otherwise, all the subproblems need to be solved for the solution to hold. The refinement is stored so that, on a successful solution, we can construct the term solving the top-level problem from the solution collection.

The `search` algorithm starts by refining the problem into new problem collections. Refining is done by tactics. Tactics are essentially just a way of organizing possible refinements. An example tactic that attempts to solve the problem by filling it with locals in scope can be seen in Listing 10.

In case refining does not create any new problem collections, the base case is reached, and the problem is trivially solved (line 9 in Listing 9). When there are new problem collections, we try to solve *any* of them. In case we cannot solve any of the problem collections, then the problem is unsolvable, and we give up by returning `Nothing` (line 15). Otherwise, we return the solutions we found.

We solve problem collections by using the `searchColl` function. Problem collections where we can't solve all the problems cannot be turned into solution collections as there is no way to build a well-formed term with problems remaining in it. We only care about cases where we can fully solve the problem, so we discard them by returning `Nothing`. On line 14 of Listing 9 we filter out unsuccessful solutions.

For successful solution collections, we substitute the refinements we took into the problem to get back the solution. The solution is a well-formed term with no remaining subproblems, which we can return to the callee.

```haskell
newtype ProbColl = (Refinement, [Problem])
newtype SolColl  = (Refinement, [Solution])

-- Find solutions to a problem
search :: Problem -> Maybe [Solution]
search p =
  case (createRefs p) of
    -- ↓ No new problems, trivially solved
    [] -> Just [TrivialSolution]
    -- ↓ Refinement created at least one subproblem
    subproblems ->
      -- Recursively solve subproblems; discard solution
      -- collections that are not fully solved.
      case (dropUnsolved $ map searchColl subproblems) of
        [] -> Nothing
        sols -> Just $ map (substitute p) sols
  where
    dropUnsolved :: [Maybe [SolColl]] -> [SolColl]
    dropUnsolved = flatten . catMaybes

-- Find a solution to every problem in problem collection
searchColl :: ProbColl -> Maybe [SolColl]
searchColl = sequence $ fmap search

-- Create refinements for problem
createRefs :: Problem -> [ProbColl]
createRefs p = flatten [tactic1 p, tactic2 p, tacticN p]

-- Create a solution to a problem from a refinement
-- and solutions to subproblems.
substitute :: Problem -> SolColl -> Solution
substitute = {- elided -}
```

Listing 9: A high-level overview of the term search algorithm used in Agsy

An example of a tactic can be seen in Listing 10.

```haskell
-- Suggest locals for solving any problem
tacticLocal :: Problem -> [ProbColl]
tacticLocal p =
  let locals = localsInScope p
  in
    map (\l -> (Refinement::SubstituteLocal p l, [])) $
    filter (\l -> couldUnify p l) locals
```

Listing 10: An example tactic that attempts to solve the problem by using locals in scope

As described above, the algorithm is built around DFS. However, the authors of [6] note that while the performance of the tool is good enough to be useful, it performs poorly on larger problems. They suggest that more advanced search space reduction techniques can be used, as well as writing it in a language that does not suffer from automatic memory management. It is also noted that there seem to be many false subproblems that can never be solved, so they suggest a parallel algorithm that could potentially prove the uselessness of those subproblems and reduce the search space.

Mimer

Mimer [7] is another proof-assistant tool for Agda that attempts to address some of the shortcomings in Agsy. As of February 2024, Mimer has become part of Agda[1] and will be released as a replacement for Agsy. According to its authors, it is designed to handle many small synthesis problems rather than complex ones. Mimer is less powerful than Agsy as it doesn't perform case splits. On the other hand, it is designed to be more robust. Other than not using case splits, the main algorithm follows the one used in Agsy and described in Section 2.2.1.1.

The main differences from the original Agsy implementation are:
1. Mimer uses memoization to avoid searching for the same term multiple times.
2. Mimer guides the search with branch costs.

Branch costs are a heuristic to hopefully guide the search to an answer faster than randomly picking branches. Mimer gives lower costs to branches that contain more local variables and fewer external definitions. The rationale for that is that it is more likely that the user wishes to use variables from the local scope than from outside of it. However, they noted that the costs of the tactics need to be tweaked in future work, as this was not their focus.

2.2.2 Term search in Standard ML

As a part of the RedPRL[2] [8] project, [9] implements term search for Standard ML.

The algorithm suggested in [9] keeps track of the subproblems in a telescope [10]. A telescope is a list of types with dependencies between them. It is a convenient data structure to keep the proof state for dependently typed languages. However, for languages without dependent types (this also includes Rust), they suggest using a regular list instead.

To more effectively propagate substitutions to subproblems in the telescope [9] suggests using BFS instead of DFS. The idea is to run all the tactics once on each subproblem, repeatedly. This way, substitutions propagate along the telescope of subproblems after every iteration. In the case of DFS, we would propagate the constraints only after exhausting the search on the first subproblem in the sequence. To better understand the difference between the BFS approach suggested and the DFS approach, let's see how each of them works.

First, let's consider the DFS approach as a baseline. The high-level algorithm for DFS is to first generate possible ways to refine the problem into new subproblems and then solve each of the subproblems fully before continuing to the next subproblem. In the snippet below, tactics create problem collections that are options we can take to refine the problem into new subproblems. After that, we attempt to solve each set of subproblems to find the first problem collection where we manage to solve all the subproblems. That problem collection effectively becomes our solution. In Listing 11 we can see that the DFS fits functional style very well, as for all the subproblems, we can just recursively call the same `solve` function again. Note that in the listing, the constraints are propagated to the remaining problems only after the problem is fully solved.

[1] Agda GitHub pull request, Mimer: a drop-in replacement for Agsy, Accessed: 2024-04-06, URL: https://github.com/agda/agda/pull/6410

[2] The red* family of proof assistants, Accessed: 2024-04-06, URL: https://redprl.org/

```haskell
solve :: Problem -> Maybe Solution
solve problem =
  let
    pcs: [ProblemCollection] = tactics problem  -- Generate possible refinements
  in
    -- Find only the first solution
    head [combineToSolution x | Just x <- map solveDFS pcs]

solveDFS :: ProblemCollection -> Maybe SolutionCollection
solveDFS [] = Just [] -- No subproblems => Empty solution collection
solveDFS (p:ps) = do
  sol <- solve p                        -- Return `Nothing` for no solution
  ps' <- propagateConstraints sol ps  -- Propagate constraints
  rest <- solvesolveDFS ps'           -- Attempt to solve other subproblems
  return sol : rest
```

Listing 11: Pseudocode for DFS search

Now let's look at how the BFS algorithm suggested in [9] works. The high-level algorithm for BFS is to generate possible ways to refine the problem into new subproblems and then incrementally solve all the subproblems in parallel. The pseudocode for it can be seen in Listing 12.

The algorithm starts by converting the given problem to a singleton problem collection. Now the produced collection is fed into `solveBFS` function, which starts incrementally solving the problem collections. In this example, we are using a queue to keep track of the problem collections we are solving. Internally, the `solveBFS` function loops over the elements of the queue until either a solution is found or the queue becomes empty. In the snippet, we check the status of the problem collection with a `status` function that tells us the status of the problem collection. The status is either:

- **AllSolved** for problem collections that do not have any unresolved subproblems in them and are ready to be converted into solutions.
- **NoSolution** for problem collections that have remaining unresolved subproblems that we are unable to make any progress on.
- **RemainingProblems** for all the problem collections that we can make progress on by incrementally stepping the problem further.

In the case of `AllSolved` we return the solution as we are done with the search. In the case of `NoSolution` we discard the problem from the queue. Otherwise, (in the case of `RemainingProblems`) we step the problem collection at the head of the queue and push the results back to the back of the queue. Now we are ready to keep iterating the loop again with the new problem collection at the head of the queue.

Stepping the problem collection steps (or adds atomic refinements) to all problems in the problem collection and propagates the constraints to the rest of the subproblems if refinements produce any new constraints. As the problem can generally be refined in multiple ways, the function returns a list of problem collections that are all possible successors to the input problem collection. Propagating the constraints is done in the `propagateConstraints` function. The function adds new constraints arising from the head element refinements to all subproblems in the problem collection.

```haskell
solve :: Problem -> Maybe Solution
solve problem =
  let
    pcs: [ProblemCollection] = toSingletonCollection problem
  in
    fmap combineToSolution (solveBFS pcs) -- Find the first solution

solveBFS :: [ProblemCollection] -> Maybe SolutionCollection
solveBFS pcs =
    loop (toQueue pcs)
  where
    loop :: Queue ProblemCollection -> Maybe SolutionCollection
    loop [] = Nothing -- Empty queue means we didn't manage to find a solution
    loop (pc:queue) = do
      case status pc of
        AllSolved         -> return toSolutionCollection ps' -- Solution found
        NoSolution        -> loop queue        -- Unable to solve, discard
        RemainingProblems ->                   -- Keep iteratively solving
          let
            pcs: [ProblemCollection] = step pc
            queue': Queue ProblemCollection = append queue pcs
          in
            loop queue'

step :: ProblemCollection -> [ProblemCollection]
step [] = []
step (p:ps) =
  let
    pcs: [ProblemCollection] = tactics p  -- Possible ways to refine head
  in
    -- Propagate constraints and step other goals
    flatten . map (\pc ->
        step $ propagateContstraints ps (extractConstraints pc)) pcs

propagateConstraints :: ProblemCollection -> Constraints -> ProblemCollection
propagateConstraints ps constraints = fmap (addConstraints constraints) ps
```

Listing 12: Pseudocode for BFS search

Consider the example where we are searching for a goal `?goal :: ([a], a -> String)` that is a pair of a list of some type and a function of that type to `String`. Similar goals in everyday life could arise from finding a list together with a function that can map the elements to strings to print them (`show` function).

Note that in this example, we want the first member of the pair to be a list, but we do not care about the types inside the list. The only requirement is that the second member of the pair can map the same type to `String`. We have the following items in scope:

```
bar : Bar
mk_foo : Bar -> Foo
mk_list_foo : Foo -> [Foo]
mk_list_bar : Bar -> [Bar]
show_bar  : Bar -> String
```

To simplify the notation, we name the goals as `?<number>`, for example, `?1` for goal 1.

First, we can split the goal of finding a pair into two subgoals: `[?1 : [a], ?2 : a -> String]`. This is the same step for BFS and DFS, as there is not much else to do with `?goal` as there

are now functions that take us to a pair of any two types except using the pair constructor. At this point, we have two subgoals to solve

```
(?1 : [a], ?2 : a -> String)
```

Now we are at the point where the differences between DFS and BFS start playing out. First, let's look at how the DFS would handle the goals. We start by focusing on `?1`. We can use `mk_list_foo` to transform the goal into finding something of the type `Foo`. Now we have the following solution and goals:

```
(mk_list_foo(?3 : Foo), ?2 : a -> String)
```

Note that although the `a` in `?s2` has to be of type `Foo`, we have not propagated this knowledge there yet as we are focusing on `?3`. We only propagate the constraints when we discard the hole as filled. We use `mk_foo` to create a new goal `?4 : Bar` which we solve by providing `bar`. Now we propagate the constraints to the remaining subgoals, `?2` in this example. This means that the second subgoal becomes `?2 : Foo -> String` as shown below.

```
(mk_list_foo(mk_foo(?4 : Bar), ?2 : a -> String)
(mk_list_foo(mk_foo(bar)), ?2 : Foo -> String)
```

However, we cannot find anything of type `Foo -> String` so we have to revert to `?1`. This time we use `mk_list_bar` to fill `?1` meaning that the remaining subgoal becomes `?2 : Bar -> String`. We can fill it by providing `show_bar`. As no more subgoals are remaining the problem is solved with the steps shown below.

```
(mk_list_bar(?3 : Bar), ?2 : a -> String)
(mk_list_bar(bar), ?2 : Bar -> String)
(mk_list_bar(bar), show_bar)
```

An overview of all the steps we took can be seen in the Listing 13.

```
1 ?goal : ([a], a -> String)
2 (?1 : [a], ?2 : a -> String)
3 (mk_list_foo(?3 : Foo), ?2 : a -> String)
4 (mk_list_foo(mk_foo(?4 : Bar), ?2 : a -> String)
5 (mk_list_foo(mk_foo(bar)), ?2 : Foo -> String) -- Revert to ?1
6 (mk_list_bar(?3 : Bar), ?2 : a -> String)
7 (mk_list_bar(bar), ?2 : Bar -> String)
8 (mk_list_bar(bar), show_bar)
```

Listing 13: DFS algorithm steps

Now let's take a look at the algorithm that uses BFS to handle the goals. The first iteration is the same as described above, after which we have two subgoals to fill.

```
(?1 : [a], ?2 : a -> String)
queue = [[?1, ?2]]
```

As we are always working on the head element of the queue, we are still working on `?1`. Once again, we use `mk_list_foo` to transform the first subgoal to `?3 : Foo`, but this time we also insert another problem collection into the queue, where we use `mk_list_bar` instead. We also propagate the information to other subgoals so that we constrain `?2` to either `Foo -> String` or `Bar -> String`.

```
(mk_list_foo(?3 : Foo), ?2 : Foo -> String)
(mk_list_bar(?4 : Bar), ?2 : Bar -> String)

queue = [[?3, ?2], [?4, ?2]]
```

In the next step, we search for something of type `Foo` for `?3` and a function of type `Foo -> String` in `?2`. We find `bar` for the first goal, but not anything for the second goal. This means we discard the branch as we are not able to solve the problem collection. Note that at this point we still have `?4` pending, meaning we have not yet exhausted the search in the current "branch". Reverting now means that we save some work that was guaranteed to not affect the overall outcome. The search space becomes

```
(mk_list_foo(mk_foo(?4 : Bar)), ?2 : <impossible>) -- discarded
(mk_list_bar(?4 : Bar), ?2 : Bar -> String)

queue = [[?4, ?2]]
```

Now we focus on the other problem collection. In this iteration, we find solutions for both of the goals. As all the problems in the problem collection get solved, we can turn the problem collection into a solution and return it.

```
(mk_list_bar(?5 : Bar), ?2 : Bar -> String)
(mk_list_bar(bar), show_bar)
```

An overview of all the steps we took can be seen in Listing 14. Note that from line 3 to line 5, there are two parallel branches, and the order between branches is arbitrary.

```
1 ?goal : ([a], a -> String)
2 (?1 : [a], ?2 : a -> String)
3 (mk_list_foo(?3 : Foo), ?2 : Foo -> String)        -- Branch 1
4 (mk_list_foo(mk_foo(?4 : Bar)), ?2 : <impossible>) -- Discard branch 1
5 (mk_list_bar(?5 : Bar), ?2 : Bar -> String)        -- Branch 2
6 (mk_list_bar(bar), show_bar)
```

Listing 14: BFS algorithm steps

In the example above, we see that BFS and propagating constraints to other subgoals can help us cut some search branches to speed up the search. However, this is not always the case. BFS is faster only if we manage to cut the proof before exhausting the search for the current goal. In case the first goal we focus on cannot be filled, DFS is faster as it doesn't do any work on filling other goals.

2.2.3 Term search in Haskell

Wingman[1] is a plugin for Haskell Language Server that provides term search. For term search, Wingman uses a library called Refinery[2] that is also based on [9] similarly to the Standard ML tool we described in Section 2.2.2.

As we described the core ideas in Section 2.2.2, we won't cover them here. However, we will take a look at some implementation details.

[1]Hackage, Wingman plugin for Haskell Language Server, Accessed: 2024-04-06, URL: https://hackage.haskell.org/package/hls-tactics-plugin

[2]Github Refinery repository, Accessed: 2024-04-06, URL: https://github.com/TOTBWF/refinery

The most interesting implementation detail for us is how BFS is achieved. Refinery uses the interleaving of subgoals generated by each tactic to get the desired effect. Let's look at the example to get a better idea of what is going on. Suppose that at some point in the term search, we have three pending subgoals: `[?1, ?2, ?3]` and we have some tactic that produces two new subgoals `[?4, ?5]` when refining `?1`. The DFS way of handling it would be

```
[?1, ?2, ?3] -> tactic -> [?4, ?5, ?2, ?3]
```

However, with interleaving, the goals are ordered in the following way:

```
[?1, ?2, ?3] -> tactic -> [?2, ?4, ?3, ?5]
```

Note that there is also a way to insert the new goals at the back of the goals list, which is the BFS way.

```
[?1, ?2, ?3] -> tactic -> [?2, ?3, ?4, ?5]
```

However, in Refinery, they have decided to go with interleaving, as it works well with tactics that produce infinite amounts of new holes due to not making any new process. Note that this works especially well because of the lazy evaluation in Haskell. In the case of eager evaluation, the execution would still halt, producing all the subgoals, so interlining would now have an effect.

2.2.4 Term search in Idris2

Idris2 [11] is a dependently typed programming language that has term search built into its compiler. Internally, the compiler makes use of a small language they call TT. TT is a dependently typed λ-calculus with inductive families and pattern-matching definitions. The language is kept as small as reasonably possible to make working with it easier.

As the term search algorithm also works on TT, we will take a closer look at it. More precisely, we will look at what they call TT_{dev}, which is TT extended with hole and guess bindings. The guess binding is similar to a let binding but without any reduction rules for guess bindings. Using binders to represent holes is useful in a dependently typed setting since one value may determine another. Attaching a guess (a generated term) to a binder ensures that instantiating one such variable also instantiates all of its dependencies.

TT_{dev} consists of terms, bindings and constants as shown in Listing 15.

```
Terms, t ::= c (constant)
    | x (variable)
    | b. t (binding)
    | t t (application)
    | T (type constructor)
    | D (data constructor)

Binders, b ::= λ x : t (abstraction)
    | let x -> t : t (let binding)
    | ∀x : t (function space)
    | ?x : t (hole binding)
    | ?x ≈ t : t (guess)

Constants, c ::= Type (type universes)
    | i (integer literal)
    | str (string literal)
```

Listing 15: TT_{dev} syntax, following [11, Fig. 1 & Fig. 6] / © Cambridge University Press 2013

Idris2 makes use of a priority queue of holes and guess binders to keep track of subgoals to fill. The goal is considered filled once the queue becomes empty.

In the implementation, the proof state is captured in an elaboration monad, which is a state monad with exceptions. The general flow of the algorithm is the following:
1. Create a new proof state.
2. Run a series of tactics to build up the term.
3. Recheck the generated term.

The proof state contains the context of the problem (local and global bindings), the proof term, unsolved unification problems, and the holes queue. The main parts of the state that change during the proof search are the holes queue and sometimes the unsolved unification problems. The holes queue changes as we try to empty it by filling all the holes. Unsolved unification problems only change if new information about unification becomes available when instantiating terms in the proof search. For example, we may have a unification problem `Unify(f x, Int)` that cannot be solved without new information. Only when we provide some concrete `f` or `x` the problem can be solved further.

Tactics in Idris2 operate on the sub-goal given by the hole at the head of the hole queue in the proof state. All tactics run relative to context, which contains all the bindings in scope. They take a term (that is, hole or guess binding) and produce a new term that is of suitable type. Tactics are also allowed to have side effects that modify the proof state.

Next, let's take a look at the primitive building blocks that are used by tactics to create and fill holes.

Operation `claim` is used to create new holes in the context of a current hole. The operation creates a new hole binding to the head of the hole queue. Note that the binding is what associates the generated hole with the current hole.

Operation `fill` is used to fill a goal with value. Given the value `v`, the operation attempts to solve the hole by creating a guess binder with `v`. It also tries to fulfill other goals by attempting to unify `v` with the types of holes. Note that the `fill` operation does not destroy the hole yet, as the guess binding it created is allowed to have more holes in it.

To destroy holes, the operation `solve` is used. It operates on guess bindings and checks if they contain any more holes. If they don't, then the hole is destroyed and substituted with the value from the guess binder.

The two-step process, with `fill` followed by `solve`, allows the elaborator to work safely with incomplete terms. This way, incomplete terms do not affect other holes (by adding extra constraints) until we know we can solve them. Once a term is complete in a guess binding, it may be substituted into the scope of the binding safely. In each of these tactics, if any step fails, or the term in focus does not solve the problem, the entire tactic fails. This means that it roughly follows the DFS approach described in Section 2.2.2.

2.2.5 Term search in Elm with Smyth

Smyth[1] is a system for program sketching in a typed functional language, approximately Elm. In [12], they describe that it uses evaluation of ordinary assertions that give rise to input-output examples, which are then used to guide the search to complete the holes. Symth uses type- and example-directed synthesis as opposed to other tools in Agda only using type-guided search for terms. The general idea is to search for terms that satisfy the goal type as well as example outputs for the term given in assertions. It is also based on a DFS but is optimized for maximal use memoization. The idea is to maximize the number of terms that have the same typing environment and can therefore be reused. This is done by factorizing the terms into smaller terms that carry less context with them. Smyth has many other optimizations, but they focus on using the information from examples and are therefore not interesting for us as they focus on optimizing the handling of data provided by examples.

2.3 Program synthesis in Rust

RusSol is a proof-of-concept tool to synthesize Rust programs from both function declarations and pre- and post-conditions. It is based on separation logic as described in [13], and it is the first synthesizer for Rust code from functional correctness specifications. Internally, it uses SuSLik's [14] general-purpose proof search framework. RusSol itself is implemented as an extension to `rustc`, the official rust compiler. It has a separate command-line tool, but internally, it reuses many parts of the compiler. Although the main use case for RusSol is quite different from our use case, they share a lot of common ground.

The idea of the tool is to specify the function declaration as follows and then run the tool on it to synthesize the program to replace the `todo!()` macro on line 5 in Listing 16.

```
1 #[requires(x < 100)]
2 #[ensures(y && result == Option::Some(x))]
3 #[ensures(!y && result == Option::None)]
4 fn foo(x: &i32, y: bool) -> Option<i32> {
5   todo!()
6 }
```

Listing 16: RusSol input program

From the preconditions (`requires` macro) and post-conditions (`ensures` macro), it can synthesize the body of the function. For the example in Listing 16, the output is shown in Listing 17.

```
match y {
  true => Some(x),
  false => None
}
```

Listing 17: RusSol output for `todo!()` macro

It can also insert `unreachable!()` macros in places that are never reached during program execution.

RusSol works at the HIR level of abstraction. It translates the information from HIR to separation logic rules that SuSLik can understand and feeds them into it. After getting a successful response, it turns the response back into Rust code, as shown in Figure 3.

[1]Smyth, Accessed: 2024-04-06, URL: https://uchicago-pl.github.io/smyth/

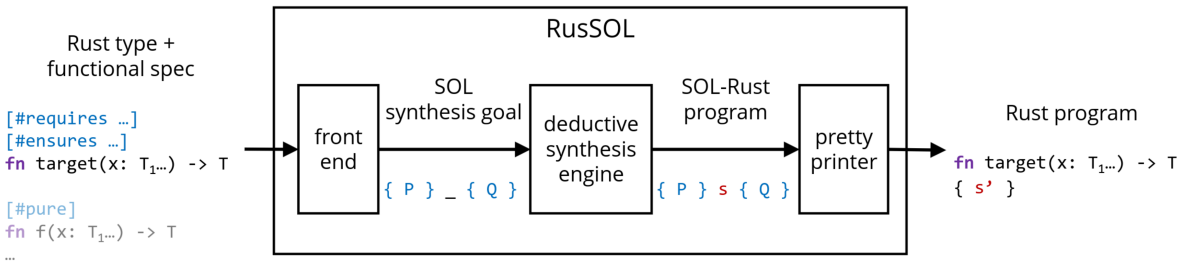

Figure 3: "RusSol workflow" by [13] / CC BY 4.0

All the programs synthesized by RusSol are guaranteed to be correct by construction. This is achieved by extracting the programs from separation logic derivations. However, in [13], they noted that they cannot prove the correctness of separation logic rules for Rust as, at this point, Rust lacks formal specification. Nevertheless, the tool was tested on 100 different crates and managed to always produce a valid code.

As the tool uses an external engine to synthesize the programs, we will not dive into its inner workings. However, we will take a look at the notes by the authors of [13], as they are very relevant to us.

The authors found that quite often the types are descriptive enough to produce useful programs, and the preconditions and postconditions are not required. This aligns with our intuition that synthesizing terms from types can be useful in practice.

The authors of RusSol pointed out the main limitations of the tool, which are:
1. It does not support traits.
2. It does not support conditionals as it lacks branch abduction.
3. It does not synthesize arithmetic expressions.
4. It does not support `unsafe` code.

They also noted that the first three of them can be solved with some extra engineering effort, but the last one requires more fundamental changes to the tool.

From the benchmarks on the top 100 crates on crates.io, it was measured that it takes about 0.5s on average to synthesize non-primitive expressions. Quite often, the synthesis time was 2-3s and sometimes reached as high as 18.3s. This is fast enough to use for filling holes, but too slow to use for autocompletion.

2.4 Autocompletion

Autocompletion is predicting what the user is typing and then suggesting the predictions to the user. In the case of programming the suggestions are usually derived from context and may be just a word from the current buffer or maybe functions reachable in the current context. It is nowadays considered one of the basic features that any integrated development environment (IDE) has built-in.

We will explore the LSP protocol in Section 2.4.2 to have a basic understanding of the constraints and features of the framework we are working in. This is essential to later understand some of our design choices for implementation, as described in Section 3.

Let's take a look at some of the popular autocompletion tools and their autocompletion-related features to get some intuition of what the common approach is for implementing them. We will be mostly looking at the kind of semantic information the tools used to provide suggestions.

Clangd

Clangd[1] is a popular autocompletion tool for C/C++. It is a language server extension to the Clang compiler and, therefore, can be used in many editors. It suggests functions, methods, variables, etc. are available in the context, and it can handle some mistyping and abbreviations of some words. For example, using a snake case instead of a camel case still yields suggestions.

For method calls, it does understand the receiver type and only suggests methods and fields that exist for the type. However, it does not try to infer the expected type of expression that is being completed and therefore is unable to prioritize methods based on that. All in all, it serves as a great example of an autocompletion tool that has a semantic understanding of the program but does not provide any functionality beyond the basics.

Pyright

Pyright[2] is a popular language server for Python. It suggests all the items that are available in scope for autocompletion, and it also suggests the methods and fields that are on the receiver type.

While it tries to provide more advanced features than `clangd`, it does not get much further due to Python being a dynamically typed language. There simply isn't that much information available before running the program. This seems to be a general limitation of all Python autocompletion tools.

Intellij

Intellij[3] is an IDE by JetBrains for Java. Similarly to all other JetBrains products, it does not use LSP but rather has all the tooling built into the product. It provides the completion of all the items in scope as well as the methods and fields of receiver type. They call them "basic completions". The tool also has an understanding of expected types, so it attempts to order the suggestions based on their types. This means that suggestions with the expected type appear first in the list.

In addition to "basic completion", they provide "type-matching completions", which are very similar to basic completion but filter out all the results that do not have matching types. There is also what they call "chain completion", which expands the list to also suggest chained method calls. Together with filtering out only matching types, it gives similar results to what term search provides. However, as this is implemented differently, its depth is limited to two, which makes it less useful. It also doesn't attempt to automatically fill all the arguments, so it works best with functions that take no arguments. For Java, it is quite useful nonetheless, as there are a lot of getter functions.

In some sense, the depth limit of two (or three together with the receiver type) is mainly a technical limitation, but it is also caused by Java using interfaces in a different way than what Rust does with traits. Interfaces in Java are meant to hide the internal representation of classes, which in some cases limits what we can provide just based on types. For example, if we are

[1] Clangd, what is clangd?, Accessed: 2024-04-06, URL: https://clangd.llvm.org/
[2] GitHub pyright repository, Accessed: 2024-04-06, URL: https://github.com/microsoft/pyright
[3] IntelliJ IDEA, Accessed: 2024-04-06, URL: https://www.jetbrains.com/idea/

expected to give something that implements `List`, we cannot prefer `ArrayList` to `LinkedList` just based on types. More common usage of static dispatch in Rust means that we more often know the concrete type and therefore can also provide more accurate suggestions based on it. In Java, there is often not enough information to suggest longer chains, as there are likely too many irrelevant suggestions.

Rust-analyzer

Rust-analyzer[1] is an implementation of the Language Server Protocol for the Rust programming language. It provides features like completion and go-to definition/references, smart refactoring, etc. This is also the tool we are extending with term search functionality.

Rust-analyzer provides all the "basic completions" that IntelliJ provides and also supports ordering suggestions by type. However, it does not support method chains, so in that regard, it is less powerful than IntelliJ for Java. Filtering by type is also not part of it, but as it gathers all the information to do it, it can be implemented rather trivially.

Other than autocompletion, it does have an interesting concept of typed holes. They are `_` (underscore) characters at expression positions that cause the program to be rejected by the compiler. Rust-analyzer treats them as holes in the program that are supposed to become terms of the correct type to make the program valid. Based on that concept, it suggests filling them with variables in scope, which is very similar to what term search does. However, it only suggests trivial ways of filling holes, so we are looking to improve on it a lot.

2.4.2 Language Server Protocol

Implementing autocompletion for every language and for every IDE results in a $O(N * M)$ complexity where N is the number of languages supported and M is the number of IDEs supported. In other words, one would have to write a tool for every language-IDE pair. This problem is very similar to the problem of compiler design with N languages and M target architectures. The problem can be reduced from $O(N * M)$ to $O(N + M)$ by separating the compiler to the front- and backend [15, Section 1.3]. The idea is that there is a unique front end for every language that lowers the language-specific constructs to an intermediate representation that is a common interface for all of them. To get machine code out of the intermediate representation, there is also a unique back end for every target architecture.

Similar ideas can also be used in building language tools. Language server protocol (LSP) has been invented to do exactly that. The Language Server Protocol[2] (LSP) is an open, JSON-RPC-based[3] protocol for use between editors and servers that provide language-specific tools for a programming language. The protocol takes the position of intermediate representation: frontends are the LSP clients in IDEs, and the backends are LSP servers. We will refer to LSP clients as just clients and LSP servers as just servers. As the protocol is standardized, every client knows how to work with any server. LSP was first introduced to the public in 2016, and now many[4] modern IDEs support it.

Some of the common functionalities provided by LSP servers include [16]:

- Go to definition/reference

[1] rust-analyzer, Accessed: 2024-04-06, URL: https://rust-analyzer.github.io/

[2] Language Server Protocol, Accessed: 2024-04-06, URL: https://microsoft.github.io/language-server-protocol/

[3] JSON-RPC 2.0 Specification, Accessed: 2024-04-06, URL: https://www.jsonrpc.org/specification

[4] The Language Server Protocol implementations: Tools supporting the LSP, Accessed: 2024-04-06, URL: https://microsoft.github.io/language-server-protocol/implementors/tools/

- Hover
- Diagnostics (warnings/errors)
- Autocompletion
- Formatting
- Refactoring routines (extract function, etc.)
- Semantic syntax highlighting

Note that the functionalities are optional, and the language server can choose which to provide.

The high-level communication between client and server is shown in Figure 4. The idea is that when the programmer works in the IDE, the client sends all text edits to the server. The server can then process the updates and send new autocompletion suggestions, syntax highlighting and diagnostics back to the client so that it can update the information in the IDE.

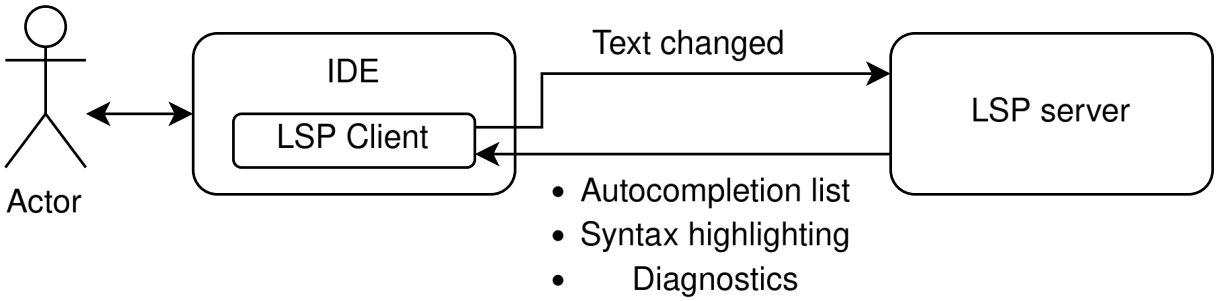

Figure 4: LSP client notifies the server of changes and user requests. The server responds by providing different functionalities to the client.

The important thing to note here is that the client starts the server the first time it requires data from it. After that, the server runs as a daemon process, usually until the editor is closed or until the client commands it to shut down. As it doesn't get restarted very often, it can keep the state in memory, which allows responding to client events faster. It is quite common that the server does semantic analysis fully only once and later only runs the analysis again for files that have changed. Caching the state and incrementally updating it is quite important, as the full analysis can take up to a considerable amount of time, which is not an acceptable latency for autocompletion nor for other operations servers provide. Caching the abstract syntax tree is a common performance optimization strategy for servers [16].

2.5 Autocompletion using machine learning

In this chapter, we will take a look at machine-learning-based autocompletion tools. As this is a very active field of development and we are not competing against it, we will not dive into how well the models perform but rather look at what the models generally do. The main focus is to see how they differ from the analytical approach we are taking with term search.

One of the use cases for machine learning is to order the suggestions [17]. Using a model for ordering the suggestions is especially useful in dynamically typed languages as it is otherwise rather hard to order suggestions. Although the Rust language has a strong type system, we still suffer from prioritizing different terms that have the same type.

In addition to ordering the analytically created suggestions, machine learning models can be used to generate code itself. Such models generate code in a variety of programming languages ([18]). The general flow is that first, the user writes the function signature and maybe some

human-readable documentation, and then prompts the model to generate the body of the function. This is very different from ordering suggestions, as the suggested code usually has many tokens, whereas the classical approach is usually limited to one or sometimes very few tokens. This is also different from what we are doing with the term search: we only try to produce code that contributes towards the parent term of the correct type. However, language models can also generate code where term search fails. Let's look at the example for the `ripgrep`[1] crate shown in Listing 18.

```
// Inside `printer_json` at `/crates/core/flags/hiargs.rs`

fn printer_json<W: std::io::Write>(&self, wtr: W) -> JSON<W> {
    JSONBuilder::new()                 // () -> JSONBuilder
        .pretty(false)                 // JSONBuilder -> JSONBuilder
        .max_matches(self.max_count)   // JSONBuilder -> JSONBuilder
        .always_begin_end(false)       // JSONBuilder -> JSONBuilder
        .build(wtr)                    // JSONBuilder -> JSON
}
```

Listing 18: Builder pattern in Rust. Setter methods return a value of the receiver type.

The type of term only changes on the first and last lines of the function body. As the lines in the middle do not affect the type of the builder in any way, there is also no way for the term search to generate them. Machine learning models, however, are not affected by this, as it may be possible to derive those lines from the function documentation, name, or rest of the context.

Although machine learning models can generate more complex code, they also have the downside of having lots of uncertainty in them. It is very hard, if not impossible, for any human to understand what the outputs are for any given input. In the context of code generation for autocompletion, this results in unexpected suggestions that may even not compile. These issues are usually addressed by filtering out syntactically invalid responses or working at the level of an abstract syntax tree, as they did in [17]. However, neither of those accounts for type nor borrow checking, which means that invalid programs can still be occasionally suggested.

[1] GitHub ripgrep repository, Accessed: 2024-04-06, URL: https://github.com/BurntSushi/ripgrep/blob/6ebebb2aaa9991694aed10b944cf2e8196811e1c/crates/core/flags/hiargs.rs#L584

3 Term search design

Before diving into the technical aspects of term search implementation, we will first explore it by giving examples of its usages in `rust-analyzer`. We will first take a look at using it for filling "holes" in the program and later dive into using it for autocompletion.

3.1 Filling holes

Filling holes is a common use case for term search, as we have found in Section 2.2. Timing constraints for it are not as strict as for autocompletion, yet the user certainly doesn't want to wait for a considerable amount of time.

One example of a hole in the Rust programming language is the `todo!()` macro. It is a "hole", as it denotes that there should be a program in the future, but there isn't now. These holes can be filled using a term search to search for programs that fit in the hole. All the programs generated by term search are valid, meaning that they compile.

Example usages can be found in Listing 19:

```
fn main() {
    let a: i32 = 0;  // Suppose we have a variable a in scope
    let b: i32 = todo!();  // Term search finds `a`
    let c: Option<i32> = todo!();  // Finds `Some(a)`, `Some(b)` and `None`
}
```

Listing 19: Filling `todo!()` holes

In addition to `todo!()` macro holes, `rust-analyzer` has a concept of typed holes, as we described in Section 2.4.1.4. From a term search perspective, they work in the same way as `todo!()` macros: term search needs to come up with a term of some type to fill them. The same example with typed holed instead of `todo!()` macros can be found in Listing 20.

```
fn main() {
    let a: i32 = 0;  // Suppose we have a variable a in scope
    let b: i32 = _;  // Term search finds `a`
    let c: Option<i32> = _;  // Finds `Some(a)`, `Some(b)` and `None`
}
```

Listing 20: Filling typed holes (_)

3.2 Term search for autocompletion

In addition to filling holes, term search can be used to give users "smarter" autocompletion suggestions as they are typing. The general idea is the same as for filling holes. We start by attempting to infer the expected type at the cursor. If we manage to infer the type, then we run the term search to get the suggestions, which we can then show to the user.

The main difference between using term search for autocompletion and using it to fill holes is that we have decided to disable borrow checking when generating suggestions for autocompletion. This means that not all the suggestions are valid programs and may need some modifications by the user.

The rationale for it comes from both the technical limitations of the tool and different expectations from the user. The main technical limitation is that borrow checking happens in the MIR layer of abstraction, and `rust-analyzer` (nor `rustc`) does not support lowering partial (the

user is still typing) programs to MIR. This means that borrow checking is not possible without big modifications to the algorithm. That, however, is out of the scope of this thesis.

In addition to technical limitations, there is also some motivation from a user perspective for the tool to give suggestions that do not borrow check. Commonly, the programmer has to restructure the program to satisfy the borrow checker [19]. The simplest case for it is to either move some lines around in the function or to add `.clone()` to avoid moving the value. For example, consider Listing 21 with the cursor at "`|`":

```
/// A function that takes an argument by value
fn foo(x: String) { todo!() }
/// Another function that takes an argument by value
fn bar(x: String) { todo!() }

fn main() {
  let my_string = String::new();

  foo(my_string);
  bar(my_s|); // cursor here at `|`
}
```

Listing 21: Autocompletion of moved values

The user wants to also pass `my_string` to `bar(...)`, but this does not satisfy the borrow checking rules as the value was moved to `foo(...)` on the previous line. The simplest fix for it is to change the previous line to `foo(my_string.clone())` so that the value is not moved. This, however, can only be done by the programmer, as there are other ways to solve it, for example, by making the functions take the reference instead of the value. As also described in [19], a common way to handle borrow checker errors is to write the code first and then fix the errors as they come up. Inspired by this, we believe that it is better to suggest items that make the program not borrow check than not suggest them at all. If we only suggest items that borrow check the `bar(my_string)` function call would be ruled out, as there is no way to call it without modifying the rest of the program.

3.3 Implementation

We have implemented term search as an addition to `rust-analyzer`, the official LSP server for the Rust language. To have a better understanding of the context we are working in, we will first describe the main operations that happen in `rust-analyzer` to provide autocompletion or code actions (filling holes in our use case).

When the LSP server is started, `rust-analyzer` first indexes the whole project, including its dependencies as well as the standard library. This is a rather time-consuming operation. During indexing, `rust-analyzer` lexes and parses all source files and lowers most of them to High-Level Intermediate Representation (HIR). Lowering to HIR is done to build up a symbol table, which is a table that has knowledge of all symbols (identifiers) in the project. This includes, but is not limited to, functions, traits, modules, ADTs, etc. Lowering to HIR is done lazily. For example, many function bodies are usually not lowered at this stage. One limitation of the `rust-analyzer` as of now is that it doesn't properly handle lifetimes. Explicit lifetimes are all mapped to `'static` lifetimes, and implicit lifetime bounds are ignored. This also limits our possibilities to do borrow checking as there simply isn't enough data available in the `rust-analyzer` yet. With the symbol table built up, `rust-analyzer` is ready to accept client requests.

Now an autocompletion request can be sent. Upon receiving a request that contains the cursor location in the source code, `rust-analyzer` finds the corresponding syntax node. If it is in a function body that has not yet been lowered, the lowering is done. Note that the lowering is always cached so that subsequent calls can be looked up from the table. With all the lowering done, `rust-analyzer` builds up the context of the autocompletion. The context contains the location in the abstract syntax tree, all the items in scope, package configuration (e.g. is nightly enabled) etc. If the expected type of the item under completion can be inferred, it is also available in the context. From the context different completion providers (functions) suggest possible completions that are all accumulated to a list. To add the term-search-based autocompletion, we introduce a new provider that takes in a context and produces a list of completion suggestions. Once the list is complete, it is mapped to the LSP protocol and sent back to the client.

3.3.1 Term search

The main implementation of term search is done at the HIR level of abstraction, and borrow checking queries are made at the MIR level of abstraction. The term search entry point can be found in `crates/hir/src/term_search.rs` and is named `term_search`. The most important inputs to the term search are the scope of the program we are performing the search at and the target type.

To better understand why the main algorithm is based around bidirectional BFS, we will discuss three iterations of the algorithm. First, we start with an algorithm that quite closely follows the algorithm we described in Section 2.2.1.1. Then we will see how we managed to achieve better results by using BFS instead of DFS, as suggested in Section 2.2.2. At last, we will see how the algorithm can benefit from bidirectional search.

3.3.2 First iteration: DFS

The first iteration of the algorithm follows the algorithm described in Section 2.2.1.1. The implementation for it is quite short, as the DFS method seems to naturally follow the DFS, as pointed out in Section 2.2.2. However, since our implementation does not use any caching, it is very slow. Because of the poor performance, we had to limit the search depth to 2, as a bigger depth caused the algorithm to run for a considerable amount of time. The performance can be improved by caching some of the found terms, but doing it efficiently is rather hard.

Caching the result means that once we have managed to produce a term of type `T`, we want to store it in a lookup table so that we won't have to search for it again. Storing the type the first time we find it is rather trivial, but it's not very efficient. The issue arises from the fact that there are no guarantees that the first term we come up with is the simplest. Consider the example of producing something of the type `Option<i32>`. We as humans know that the easiest way to produce a term of that type is to use the `None` constructor that takes no arguments. The algorithm, however, might first take the branch of using the `Some(...)` constructor. Now we have to also recurse to find something of type `i32`, and potentially iterate again and again if we do not have anything suitable in scope. Even worse, we might end up not finding anything suitable after fully traversing the tree we got from using the `Some(...)` constructor. Now we have to also check the `None` subtree, which means that we only benefit from the cache if we want to search for `Option<i32>` again.

This is not a problem if we want to retrieve all possible terms for the target type, however, that is not always what we want to do. We found that for bigger terms, it is better to produce a

term with new holes in it, even when we have solutions for them, just to keep the number of suggestions low. Consider the following example:

```
let var: (bool, bool) = todo!();
```

If we give the user back all possible terms, then the user has to choose between the following options:

```
(false, false)
(false, true)
(true, false)
(true, true)
```

However, we can simplify it by only suggesting the use of a tuple constructor with two new holes in it.

```
(todo!(), todo!())
```

If there are only a few possibilities to come up with a solution, then showing them all isn't a problem. However, it is quite common for constructors or functions to take multiple arguments. As the number of terms increases exponentially relative to the number of arguments a function/constructor takes, the number of suggestions grows very fast. As a result, quite often, all the results don't even fit on the screen. In Section 3.3.3, we will introduce an algorithm to handle this case. For now, it is sufficient to acknowledge that fully traversing the search space to produce all possible terms is not the desired approach, and there is some motivation to cache the easy work to avoid the hard work, not vice versa. Branch costs suggested in [7] can potentially improve this, but the issue remains as this is simply a heuristic.

Another observation from implementing the DFS algorithm is that, while most of the algorithm looked very elegant, the "struct projection" tactic described in Section 3.4.1.6 was rather awkward to implement. The issue arose with the projections having to include all the fields from the parent struct as well as from the child struct. Including only the child "leaf" fields is very elegant with DFS, but also including the intermediate fields caused some extra boilerplate.

Similar issues arose when we wanted to speed up the algorithm by running some tactics, for example, the "impl method" only on types that we have not yet run it on. Doing it with DFS is possible, but it doesn't fit the algorithm conveniently. As there were many issues with optimizing the DFS approach, we decided to not improve it further and turned to a BFS-based algorithm instead.

3.3.3 Second iteration: BFS

The second iteration of our algorithm was based on BFS, as suggested in [9]. However, it differs from it by searching in the opposite direction.

To not confuse the directions, we use *forward* when we are constructing terms from what we have (working towards the goal) and *backward* when we work backward from the goal. This aligns with the forward and the backward directions in generic path finding, where the forward direction is from source to target and the backward direction is from target to source.

The algorithm in [9] starts from the target type and starts working backward from it toward what we already have. For example, if we have a function in scope that takes us to the goal, we create new goals for all the arguments of the function, therefore we move backward from the return type towards the arguments. Our algorithm, however, works in the forward direction, meaning that we start from what we have in scope. We try to apply all the functions, etc. to then build new types from what we have and hopefully, at some point, arrive at the target type.

In [20], they argue that taking the forward (bottom-up) approach will yield speedups when the active frontier is a substantial fraction of the total graph. We believe that this might be the case for term search, as there are many ways to build new types available (functions/constructors/methods). Going in the forward direction, all the terms we create are well-formed and do not have holes in them. This means that we do not need problem collections, as there are never multiple subproblems pending that have to all be solved for some term to be well-formed. As there is a potential speedup and the implementation seems to be easier, we decided to experiment with using the forward approach.

Going in the "forward" direction also makes writing some of the tactics easier. Consider the example of struct projections. In the backward direction, we would start with the struct field and then later search if we had the struct available. This works, but it is rather hard to understand, as we usually write code for projections in the forward direction. With BFS going in the forward direction, we can just visit all the fields of struct types in every iteration, which roughly follows how we usually write code. The issue of awkward handling of fields together with their fields also goes away, as we can consider only one level of fields in every iteration. With multiple iterations, we manage to cover fields of nested structs without needing any boilerplate.

In this iteration, we also introduce the cache to the algorithm. The idea of the cache is to keep track of types we have reached so that we can query for terms of that type in $O(1)$ time complexity. Since in practice we also care about terms that unify with the type, we get the complexity of $O(n)$ where n is the number of types in the cache. This is still a lot faster than traversing the tree, as iterating the entries in the map is a quite cheap operation. With this kind of graph, we managed to increase the search depth to 3-4, depending on the size of the project.

In the DFS approach without cache, the main limitation was time complexity, but now the limitation is memory complexity. The issue is producing too many terms for a type. In Section 3.3.2, we discussed that there are often too many terms to present to the user. However, now we find that there are also too many terms to keep in memory due to their exponential growth as the depth increases. Luckily, the idea of suggesting user terms that have new holes in them also reduces the memory complexity a lot.

To avoid producing too many terms we cache terms using enum shown in Listing 22.

```
type Cache = Map<Type, AlternativeExprs>;

enum AlternativeExprs {
    /// There are few expressions, so we keep track of them all
    Few(Set<Expr>),
    /// There are too many expressions to keep track of
    Many,
}
```

Listing 22: Cache data structure for the term search algorithm

The idea is that if there are only a few terms of a given type, we keep them all so that we can provide the full term to the user. However, if there are too many of them to keep track of, we just remember that we can come up with a term for a given type, but we won't store the terms themselves. The cases of `Many` later become the holes in the generated term.

In addition to decreasing memory complexity, this also reduces time complexity a lot. Now we do not have to construct the terms if we know that there are already many of them. This can be achieved quite elegantly by using iterators in Rust. Iterators in Rust are lazy, meaning that they only do work if we consume them. In our case, consuming the iterator is extending the `AlternativeExprs` in the cache. However, if we are already in many cases, we can throw away the iterator without performing any computation. This speeds up the algorithm a lot, so now we can raise the depth of search to 10+ with it still outperforming the previous algorithms on a timescale.

The algorithm itself is quite simple. The pseudocode for it can be seen in Listing 23. We start by gathering all the items in scope to `defs`. These items include local values and constants, as well as all visible functions, constructors, etc. Next, we initialize the lookup table with the desired *many threshold* for the alternative expressions shown in Listing 22. The lookup table owns the cache, the state of the algorithm and some other values for optimizations. We will discuss the exact functionalities of the lookup table in Section 3.3.3.1.

Before entering the main loop, we populate the lookup table by running a tactic called `trivial`. Essentially, it attempts to fulfill the goal by trying the variables we have in scope. More information about the `trivial` tactic can be found at Section 3.4.1.1. All the terms it produces get added to the lookup table and can be later used in other tactics. After that, we iteratively expand the search space by attempting different tactics until we have exceeded the preconfigured search depth. During every iteration, we sequentially attempt different tactics. All tactics build new types from existing types (constructors, functions, methods, etc.) and are described in Section 3.4. The search space is expanded by adding new types to the lookup table. An example of it can be seen in Figure 5. We keep iterating after finding the first match, as there may be many terms of the given type. Otherwise, we would never get suggestions for `Option::Some(..)`, as `Option::None` usually comes first as it has fewer arguments. In the end, we filter out solutions that do not take us closer to the goal.

```rust
pub fn term_search(ctx: &TermSearchCtx) -> Vec<Expr> {
    let mut defs = ctx.scope.process_all_names(...);
    let mut lookup = LookupTable::new(ctx.many_threshold);

    // Try trivial tactic first, also populates lookup table
    let mut solutions: HashSet<Expr> =
        tactics::trivial(ctx, &defs, &mut lookup).collect();

    for _ in 0..ctx.config.depth {
        lookup.new_round();

        solutions.extend(tactics::type_constructor(ctx, &defs, &mut lookup));
        solutions.extend(tactics::free_function(ctx, &defs, &mut lookup));
        solutions.extend(tactics::impl_method(ctx, &defs, &mut lookup));
        solutions.extend(tactics::struct_projection(ctx, &defs, &mut lookup));
        solutions.extend(tactics::impl_static_method(ctx, &defs, &mut lookup));
    }

    solutions.into_iter().filter(|it| !it.is_many()).collect()
}
```

Listing 23: Forward direction term search pseudocode

As we can see from the Listing 23, we start with what we have (locals, constants, and statics) and work towards the target type. This is in the opposite direction compared to the tools we have looked at previously. To better understand how the search space is expanded, let us look at Figure 5.

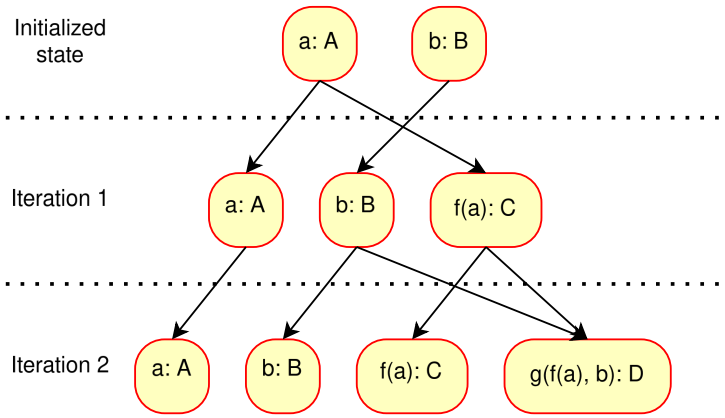

Figure 5: Iterative term search state expansion. We start with terms of types A and B. With every iteration, we keep the terms from the previous iteration and add new terms if possible.

We start with variables `a` of type `A` and `b` of type `B`. In the first iteration, we can use the function $f : A \to C$ on `a` and get something of the type `C`. in the iteration after that, we can use $g : C \times B \to D$ and produce something of type `D`.

Once we have reached the maximum depth, we take all the elements that unify with the goal type and return all paths that take us to the goal type.

Lookup table

The main task of the lookup table throughout the algorithm is to keep track of the state. The state consists of the following components:

1. *Terms reached* (grouped by type)
2. *New types reached* (since last iteration)
3. *Definitions used* and *definitions exhausted* (for example, functions applied)

4. *Types wishlist* (types that have been queried, but not reached)

Terms reached keeps track of the search space we have already covered (visited types) and allows querying terms in $O(1)$ complexity for exact type and $O(n)$ complexity for types that unify. It is important to note that it also performs the transformation of taking a reference if we query for a reference type. This is only to keep the implementation simple and the memory footprint low. Otherwise, having a separate tactic for taking a reference of this type would be preferred.

New types reached keeps track of new types added to *terms reached* so that we can iterate only over them in some tactics to speed up the execution.

Definitions used serves also only the purpose of speeding up the algorithm by avoiding definitions that have already been used.

Types wishlist keeps track of all the types we have tried to look up from terms reached but not found. They are used in the static method tactic (see Section 3.4.1.7) to only search for static methods on types we haven't reached yet. This is another optimization for speed described in the Section 3.4.1.7.

The main downside of the lookup table implementation we have is that it poorly handles types that take generics. We only store types that are normalized, meaning that we have substituted the generic parameter with some concrete type. In the case of generics, it often means that the lookup table starts growing exponentially. Consider the example of using the `Option` type.

```
Some(T) | None
Some(Some(T)) | Some(None) | Some(T) | None
Some(Some(Some(T))) | Some(Some(None)) | Some(Some(T)) | Some(None) | Some(T) | None
```

With every iteration, two new terms of a new type become available, even though it is unlikely one would ever use them. However, since `Option` takes only one generic argument, the growth is linear as many of the terms cancel out due to already being in the cache. If we have something with multiple generic parameters, it becomes exponential. Consider the example of wrapping the types we have to pair (a tuple with two elements). At first, we have n types. After the first iteration, we have n^2 new types as we are taking the Cartesian product. In the second iteration, we can create a pair by taking one of the elements from the original set of types and the second element from the set of pairs we have. As for every pair, there are n original types to choose from and we get n^3 pairs and also all the pairs of pairs. Even without considering the pairs of pairs, we see that the growth is exponential.

To keep the search space to a reasonable size, we ignore all types with generics unless they are directly related to the goal. This means that we limit the depth for the generics to 1, which is a very severe but necessary limitation. In Section 3.3.4, we will discuss how to get around this limitation.

3.3.4 Third iteration: Bidirectional BFS

The third iteration of our algorithm is a small yet powerful improvement on the second iteration described in Section 3.3.3. This iteration differs from the previous one by improving the handling of generics. We note that the handling of generics is a much smaller problem if going in the backward direction similar to other term search tools. This is because we can only construct

the types that contribute towards reaching the goal. However, if we only go in the backward direction, we can still end up with terms such as `Some(Some(...)).is_some()` that do contribute towards the goal but not in a very meaningful way. BFS copes with these kinds of terms quite well, as the easiest paths are taken first. However, with multiple iterations, many not-so-useful types get added to the lookup table nonetheless. Note that the trick with lazy evaluation of iterators does not work here as the terms have types not yet in the lookup, meaning we cannot discard them. Filtering them out in a backward direction is possible but not trivial.

To benefit from better handling of generics going in the backward direction and an otherwise more intuitive approach of going in the forward direction, we decided to make the search bidirectional. The forward direction starts with the locals we have and starts expanding the search space from there. Tactics that work in the forward direction ignore all types where we need to provide generic parameters. Other tactics start working backward from the goal. All the tactics that work backward do so to better handle generics.

Going backward is achieved by using the types wishlist component of the lookup table. We first seed the wishlist with the target type. During every iteration, the tactics working backward from the target type only work with the concrete types we have in our wishlist. For example, if there is `Option<Foo>` in the wishlist and we work with the `Option<T>` type, we know to substitute the generic type parameter `T` with `Foo`. This way, we avoid polluting the lookup table with many types that likely do not contribute towards the goal. All the tactics add types to the wishlist, so forward tactics can benefit from the backward tactics (and vice versa) before meeting in the middle. With some tactics, such as using methods on types only working in the forward direction, we can conveniently avoid adding complex types to the wishlist if we only need them to get something simple, such as `bool` in the `Some(Some(...)).is_some()` example.

3.4 Tactics

We use tactics to expand the search space for the term search algorithm. All the tactics are applied sequentially, which causes a phase-ordering problem as tactics generally depend on the results of others. However, the ordering of tactics problem can be fixed by running the algorithm for more iterations. Note that some tactics also use heuristics for performance optimization that also suffer from the phase ordering problem, but they cannot be fixed by running the algorithm for more iterations.

All the tactic function signatures follow the simplified function signature shown in Listing 24.

```
fn tactic_name(
    ctx: &TermSearchCtx,
    defs: &HashSet<ScopeDef>,
    lookup: &mut LookupTable,
) -> impl Iterator<Item = Expr>
```

Listing 24: Term search tactic signature. Arguments `ctx` and `defs` give all the available context. State is encapsulated in `lookup`. All tactics return an iterator that yields terms that unify with the goal.

All the tactics take in the context of term search, definitions in scope, and a lookup table and produce an iterator that yields expressions that unify with the goal type (provided by the context). The context encapsulates the semantics of the program, the configuration for the term search, and the goal type. Definitions are all the definitions in scope that can be used by tactics. Some examples of definitions are local variables, functions, constants, and macros. The

definitions in scope can also be derived from the context, but they are kept track of separately to speed up the execution by filtering out definitions that have already been used. Keeping track of them separately also allows querying them only once, as they do not change throughout the execution of the algorithm. The lookup table is used to keep track of the state of the term search, as described in Section 3.3.3.1. The iterator produced by tactics is allowed to have duplicates, as filtering of them is done at the end of the algorithm. We decided to filter at the end because it is hard to guarantee that different tactics do not produce the same elements, but without the guarantee of uniqueness, there would have to be another round of deduplication nevertheless.

Tactic "trivial"

A tactic called "trivial" is one of the most trivial tactics we have. It only attempts items we have in scope and does not consider any functions/constructors. The items in scope contain:

1. Constants
2. Static items
3. Generic parameters (constant generics[1])
4. Local items

As this tactic only depends on the values in scope, we don't have to call it every iteration. We only call it once before any of the other tactics to populate the lookup table for forward direction tactics with the values in scope.

Tactic "famous types"

"Famous types" is another rather trivial tactic. The idea of the tactic is to attempt values of well-known types. Those types and values are:

1. `true` and `false` of type `bool`
2. `()` of unit type `()`

While we usually try to avoid creating values out of the blue, we make an exception here. The rationale for making the types we generate depend on the types we have in scope is that usually, the programmer writes the code that depends on inputs or previous values. Suggesting something else can be considered distracting. However, we find these values to be common enough to usually be a good suggestion. Another reason is that we experienced our algorithm "cheating" around depending on values anyway. It constructed expressions like `None.is_none()` and `None.is_some()` for `true`/`false` which are valid but most likely never what the user wants. For unit types, it can use any function that has "no return type", meaning it returns a unit type. There is usually at least one that kind of function in scope, but suggesting it is unexpected more often than suggesting `()`. Moreover, suggesting a random function with a `()` return type can often be wrong as the functions can have side effects. Similarly to the tactic "trivial", this tactic helps to populate the lookup table for the forward pass tactics.

Tactic "data constructor"

"Data constructor" is the first of our tactics that takes us from terms of some types to terms of other types. The idea is to attempt to apply the data constructors of the types we have in scope. We try them by looking for terms for each of the arguments the constructor has from the lookup table. If we have terms for all the arguments, then we have successfully applied the constructor. If not, then we cannot apply the constructor at this iteration of the algorithm.

[1]The Rust Reference, Generic parameters, Accessed: 2024-04-06, URL: https://doc.rust-lang.org/reference/items/generics.html

The tactic includes both sum and product types (`enum` and `struct` for Rust).

As compound types may contain generic arguments, the tactic works in both forward and backward directions. The forward direction is used if the ADT does not have any generic parameters. The backward direction is used for types that have generic parameters. In the backward direction, all the generic type arguments are taken from the types in the wishlist. By doing that, we know that we only produce types that somehow contribute to our search.

The tactic avoids types that have unstable generic parameters that do not have default values. Unstable generics with default values are allowed, as many of the well-known types have unstable generic parameters that have default values. For example, the definition for `Vec` type in Rust is the following:

```
struct Vec<T, #[unstable] A: Allocator = Global>
```

As the users normally avoid providing generic arguments that have default values, we also decided to avoid filling them. This means that for the `Vec` type above, the algorithm only tries different types for `T` but never touches the `A` (allocator) generic argument.

Tactic "free function"
This tactic attempts to apply free functions we have in scope. It only tries functions that are not part of any `impl` block (associated with type or trait) and are therefore considered "free".

A function can be successfully applied if we have terms in the lookup table for all the arguments that the function takes. If we are missing terms for some arguments, we cannot use the function, and we try again in the next iteration when we hopefully have more terms in the lookup table.

We have decided to filter out all the functions that have non-default generic parameters. This is because `rust-analyzer` does not have proper checking for the function to be well-formed with a set of generic parameters. This is an issue if the generic parameters that the function takes are not present in the return type.

As we ignore all the functions that have non-default generic parameters we can run this tactic in only the forward direction. The tactic avoids functions that return types that contain references (Section 3.4). However, we do allow function arguments to take items by shared references as this is a common practice to pass by reference rather than value.

Tactic "impl method"
This tactic attempts functions that take a `self` parameter. This includes both trait methods and methods implemented directly on type. Examples for both of these cases are shown in Listing 25. Both of the impl blocks are highlighted, and each of them has a single method that takes a `self` parameter. These methods can be called as `example.get_number()` and `example.do_thingy()`.

```rust
struct Example {
    number: i32,
}

impl Example {
    fn get_number(&self) -> i32 {
        self.number
    }
}

trait Thingy {
    fn do_thingy(&self);
}

impl Thingy for Example {
    fn do_thingy(&self) {
        println!("doing a thing! also, number is {}!", self.number);
    }
}
```

Listing 25: Examples of `impl` blocks, highlighted in yellow

Similarly to the "free function" tactic, it also ignores functions that have non-default generic parameters defined on the function for the same reasons. However, generics defined on the `impl` block pose no issues as they are associated with the target type, and we can provide concrete values for them.

A performance tweak for this tactic is to only search the `impl` blocks for types that are new to us, meaning that they were not present in the last iteration. This implies we run this tactic only in the forward direction, i.e. we need to have a term for the receiver type before using this tactic. This is a heuristic that speeds up the algorithm quite a bit, as searching for all `impl` blocks is a costly operation. However, this optimization does suffer from the phase ordering problem. For example, we may want to use use some method from the `impl` block later, when we have reached more types and covered a type that we need for an argument of the function.

We considered also running this tactic in the reverse direction, but it turned out to be very hard to do efficiently. The main issue is that there are many `impl` blocks for generic `T` that do not work well with the types wishlist we have, as it pretty much says that all types belong to the wishlist. One example of this is shown in Listing 26.

```rust
impl<T: fmt::Display + ?Sized> ToString for T {
    fn to_string(&self) -> String { /* ... */ }
}
```

Listing 26: Blanket `impl` block for `ToString` trait in the standard library. All the types that implement `fmt::Display` also implement `ToString`.

One interesting aspect of Rust to note here is that even though we can query the `impl` blocks for type, we still have to check that the receiver argument is of the same type. This is because Rust allows also some other types that dereference to the type of `Self` for the receiver argument[1].

[1]The Rust Reference, Associated Items, Accessed: 2024-04-06, URL: https://doc.rust-lang.org/reference/items/associated-items.html#methods

These types include but are not limited to `Box<S>`, `Rc<S>`, `Arc<S>`, and `Pin<S>`. For example, there is a method signature for the `Option<T>` type in the standard library[1] shown in Listing 27.

```
impl<T> Option<T> {
    pub fn as_pin_ref(self: Pin<&Self>) -> Option<Pin<&T>> { /* ... */ }
}
```

Listing 27: Receiver argument with type other than `Self`

As we can see from the snippet above, the type of `Self` in the `impl` block is `Option<T>`. However, the type of `self` parameter in the method is `Pin<&Self>`, which means that to call the `as_pin_ref` method, we need to have an expression of type `Pin<&Self>`.

We have also decided to ignore all the methods that return the same type as the type of `self` parameter. This is because they do not take us any closer to the goal type, and we have considered it unhelpful to show users all the possible options. If we allowed them, then we would also receive expressions such as `some_i32.reverse_bits().reverse_bits().reverse_bits()` which is valid Rust code but unlikely something the user wished for. Similar issues often arise when using the builder pattern, as shown in Listing 18.

Tactic "struct projection"

"Struct projection" is a simple tactic that attempts all field accesses of struct. The tactic runs only in the forward direction, meaning we only try to access fields of the target type rather than search for structs that have fields of the target type. In a single iteration, it only goes one level deep, but with multiple iterations, we cover all the fields of substructs.

This tactic greatly benefits from the use of BFS over DFS, as the implementation for accessing all the fields of the parent struct is rather trivial, and with multiple iterations, we get the full coverage, including substruct fields. With DFS, the implementation was much more cumbersome, as simple recurring on all the fields leaves out the fields themselves. As a result, the implementation for DFS was about two times longer than the implementation for BFS.

As a performance optimization, we only run this tactic on every type once. For this tactic, this optimization does not reduce the total search space covered, as accessing the fields doesn't depend on the rest of the search space covered.

Tactic "static method"

The "Static method" tactic attempts static methods of `impl` blocks, that is, methods that are associated with either type or trait, but do not take the `self` parameter. Some examples of static methods are `Vec::new()` and `Default::default()`.

As a performance optimization, we query the `impl` block for types we have a wishlist, meaning we only go in the backward direction. This is because we figured that the most common use case for static methods is the factory method design pattern described in [21]. Querying `impl` blocks is a costly operation, so we only do it for types that are contributing towards the goal, meaning they are in the wishlist.

Similarly to the "Impl method" tactic, we ignore all the methods that have generic parameters defined at the method level for the same reasoning.

[1]Rust standard library source code, Accessed: 2024-04-06, URL: https://doc.rust-lang.org/src/core/option.rs.html#715

Tactic “make tuple”

The “make tuple” tactic attempts to build types by constructing a tuple of other types. This is another tactic that runs only in the backward direction, as otherwise, the search space would grow exponentially. In Rust, the issue is even worse as there is no limit to how many items can be in a tuple, meaning that even with only one term in scope, we can create infinitely many tuples by repeating the term an infinite number of times.

Going in the backward direction, we can only construct tuples that are useful and therefore keep the search space reasonably small.

4 Evaluation

In this chapter, we evaluate the performance of the three iterations of the algorithm as implemented in Section 3.3.1. The main focus is on the third and final iteration, but we compare it to previous iterations to highlight the differences.

First, we perform an empirical evaluation of the tree algorithms by performing a resynthesis on existing Rust programs. Later, we focus on some hand-picked examples to show the strengths and weaknesses of the tool.

4.1 Resynthesis

Resynthesis is using the tool to synthesize programs for which a reference implementation exists. This allows us to compare the generated suggestions to known-good programs. For resynthesis, proceed as follows:

1. Take an existing open-source project as a reference implementation.
2. Remove one expression from it, creating a hole in the program.
3. Run the term search in the hole.
4. Compare the generated expressions to the reference implementation.
5. Put back the original expression and repeat on the rest of the expressions.

Choice of expressions

We chose to perform resynthesis only on the *tail expressions* of blocks, as we consider this the most common use case for our tool. A block expression is a sequence of statements followed by an optional tail expression, enclosed in braces (`{...}`). For example, the body of a function is a block expression, and the function evaluates to the value of its tail expression. Block expressions also appear as the branches of `if` expressions and `match`-arms. For some examples, see Listing 28.

```
1  fn foo(x: Option<i32>) -> Option<bool> {
2    let y = {
3      /* Compute something */
4      true
5    }
6    let res = match x {
7      Some(it) => {
8        if x < 0 {
9          /* Do something */
10         true
11       } else {
12         /* Do something else */
13         false
14       }
15     }
16     None => {
17       true
18     }
19   }
20
21   Some(res)
22 }
```

Listing 28: Examples of tail expressions: in a scoping block (line 4), in branch arms (line 10, 13, 17) and the return position (line 21).

Choice of metrics

For resynthesis, we are interested in the following metrics:

1. *Holes filled* represents the fraction of tail expressions where the algorithm finds at least one term that satisfies the type system. The term may or may not be what was there originally.
2. *Holes filled (syntactic match)* represents the share of tail expressions relative to the total number of terms that are syntactically equal to what was there before. Note that syntactical equality is a very strict metric, as programs with different syntax may have the same meaning. For example, `Vec::new()` and `Vec::default()` produce exactly the same behavior. As deciding the equality of the programs is generally undecidable according to Rice's theorem [22], we will not attempt to consider the equality of the programs and settle for syntactic equality. To make the metric slightly more robust, we compare the program's ASTs, effectively removing all the formatting before comparing.
3. *Average time* represents the average time for a single term search query. Note that although the cache in term search does not persist between runs, the lowering of the program is cached. This is however also true for the average use case of `rust-analyzer` as it only wipes the cache on restart. To benchmark the implementation of term search rather than the rest of `rust-analyzer` we run term search on hot cache.
4. *Terms per hole* - This shows the average number of options provided to the user.

These metrics are relatively easy to measure and relevant to users: They tell us how often the tool offers assistance, how much of it is useful, and if it slows down the user's flow of development. All experiments are conducted on a consumer-grade computer with an AMD Ryzen 7 CPU and 32GB of RAM.

Choice of reference implementations

For our experiments, we select a number of open-source Rust libraries. In Rust, *crate* is the name for a library. We use *crates.io*[1], the Rust community's crate registry as a source of information of the most popular crates. *Crates.io* is the *de facto* standard crate registry, so we believe that it reflects the popularity of the crates in the Rust ecosystem very well.

We select representative examples of different kinds of Rust programs by picking crates from popular categories on *crates.io*. For each category containing at least 1000 crates, we select its top 5 crates, sorted by all-time downloads. This leaves us with 31 categories and a total of 155 crates. The full list of crates can be seen in Appendix 2.

Results

First, we are going to take a look at how the hyperparameter of search depth affects the chosen metrics.

We measured *holes filled* and the number of *terms per hole* for search depths up to 5 (Figure 6, Table 1). For search depth 0, only trivial tactics (Section 3.4.1.1 and Section 3.4.1.2) are run. This results in 18.9% of the holes being filled, with only 2.5% of the holes having syntactic matches. Beyond the search depth of 2, we noticed barely any improvements in the portion of holes filled. At depth 2, the algorithm fills 74.9% of holes. By doubling the depth, the number of holes filled increases only by 1.5%pt to 76.4%. More interestingly, we can see from Table 1 that syntactic matches start to decrease after a depth of 3. This is because we get more results

[1]The Rust community's crate registry, Accessed: 2024-04-06, URL: https://crates.io/

for subterms and squash them into `Many`, i.e. replace them with a new hole. Terms that would result in syntactic matches get squashed, resulting in a decrease in syntactic matches.

The number of terms per hole follows a similar pattern to holes filled, but the curve is flatter. At depth 0, we have, on average, 15.1 terms per hole. At depths above 4, this number plateaus at around 23 terms per hole. Note that a larger number of terms per hole is not always better. Too many terms might be overwhelming to the user.

Over 15 terms per hole at depth 0 is more than we expected, so we will more closely investigate the number of terms per hole in Section 4.2.1.8.

To more closely investigate the time complexity of the algorithm, we run the experiment up to a depth of 20. We estimate that running the experiment on all 155 crates would take about half a month. To speed up the process, we select only the most popular crate for each category. This results in 31 crates in total (Appendix 4).

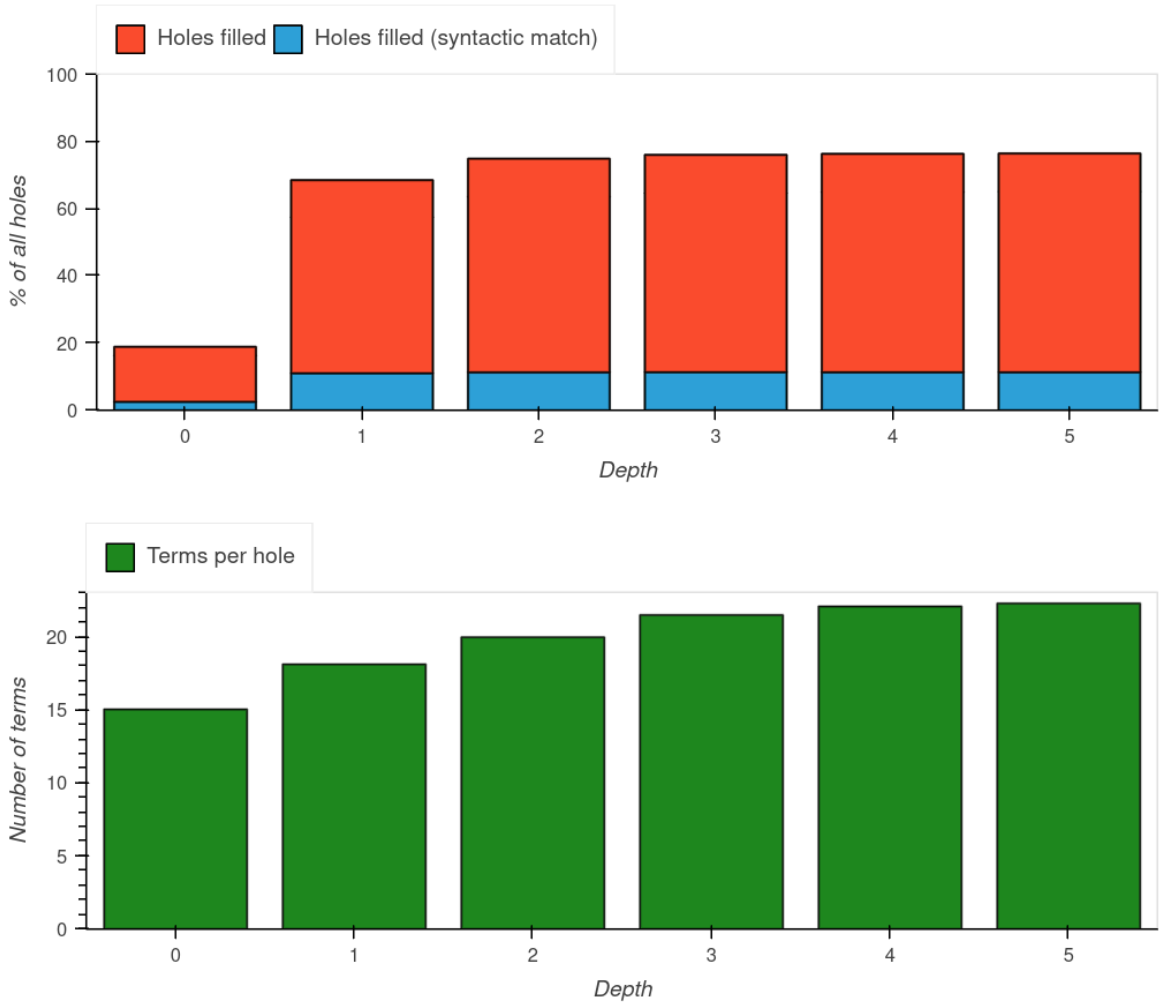

Figure 6: The effect of search depth on the fraction of holes filled, and the average number of terms per hole. For depth >2, the number of holes filled plateaus. Syntactic matches do not improve at depth above 1. The number of terms per hole starts at a high number of 15 and increases until the depth of 4 reaching 22.

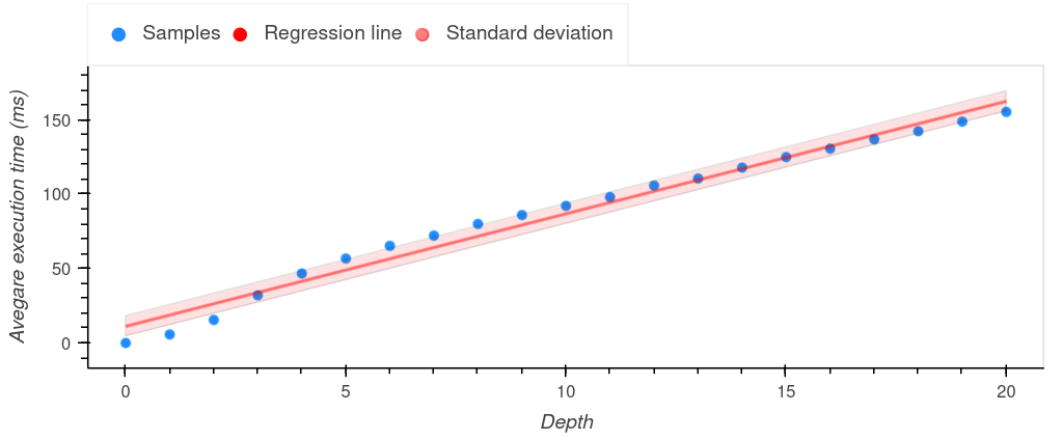

Figure 7: The execution time of the algorithm is linear in the search depth. Slope = $7.6\frac{\text{ms}}{\text{depth}}$, standard deviation = 6.7ms

We observe that in the average case, the execution time of the algorithm is in linear relation to the search depth (Figure 7). Increasing depth by one adds about 8ms of execution time on average.

We can see that increasing the search depth over two can have somewhat negative effects. The search will take longer, and there will be more terms. More terms often mean more irrelevant suggestions. By examining the fraction of holes filled and holes filled with syntactic matches, we see that both have reached a plateau at depth 2. From that, we conclude that we are mostly increasing the number of irrelevant suggestions. This can also be seen in Figure 6, where the fraction of holes filled has stalled after the second iteration, but execution time keeps increasing linearly in Figure 7.

With a depth of 2, the program manages to generate a term that satisfies the type for 74.9% of all holes. In 11.0% of searches, the generated term syntactically matches the original term. The average number of terms per hole is 20, and they are found in 49ms. However, the numbers vary a lot depending on the style of the program. The standard deviation between categories for the average number of terms is about 56 terms, and the standard deviation of the average time is 135ms. Both of these are greater than the average numbers themselves, indicating large differences between categories. We discuss the categories that push the standard deviation so high in Section 4.2.1.8.

Depth	Holes filled	Syntactic matches	Terms per hole	Average time
0	18.9%	2.5%	15.1	0.5ms
1	68.6%	11.0%	18.1	7.1ms
2	74.9%	11.3%	20.0	49.5ms
3	76.1%	11.4%	21.5	79.5ms
4	76.4%	11.3%	22.1	93.9ms
5	76.5%	11.3%	22.3	110.1ms

Table 1: Depth hyperparameter effect on metrics. *Holes filled* plateaus at 76% on depth 2. *Syntactic matches* reaches 11.4% at depth 3 and starts decreasing. *Terms per hole* starts at a high number of 15 per hole, plateaus at depth 4 around 22 terms per hole. *Average time* increases about linearly.

To give some context to the results, we decided to compare them to results from previous iterations of the algorithm. However, both of the previous algorithms were so slow with some particular crates that we couldn't run them on the whole set of benchmarks. As some of the worst cases are eliminated for iterations `v1` and `v2`, the results in Table 2 are more optimistic for `v1` and `v2` than for the final iteration of the algorithm. Nevertheless, the final iteration manages to outperform both of the previous iterations.

The first iteration performs significantly worse than others, running almost two orders of magnitude slower than other iterations and filling about only a third of the holes compared to the final iteration of the algorithm. As the performance of the first iteration is much worse than the later iterations, we will not dive into the details of it.

Instead, we compare the last two iterations more closely. The final iteration manages to fill 1.6 times more holes than the second iteration of the algorithm at depth 3. It also fills 1.8 times more holes with syntactic matches. These results were achieved in 12% less time than the second iteration.

Algorithm	**Holes filled**	**Syntactic matches**	**Terms per hole**	**Avg time**
`v1`, depth = 1	26%	4%	5.8	4900ms
`v2`, depth = 3	46%	6%	17.2	90ms
`v3`, depth = 3	76%	11%	21.5	79ms

Table 2: Comparison of algorithm iterations. `v1` performs the worst in every metric, especially execution time. `v2` runs slightly slower than `v3`, and fills significantly fewer holes.

In addition to average execution time, we care about the low latency of suggestions. We chose 100ms as a latency threshold, which we believe is low enough for responsive autocompletion. This is a recommended latency threshold for web applications ([23]), and the mean latency of writing digraphs while programming is around 170ms ([24]). We will use our algorithm with a depth of 2, as this seems to be the optimal depth for autocompletion.

We found that 87% of holes can be filled within 100ms. In 8 of the categories, all holes could be filled in 100ms. The main issues arose in the categories "hardware-support" and "external-ffi-bindings", in which only 6% and 16% of the holes could be filled within the 100ms threshold. These categories were also problematic from the other aspects, and we will discuss the issues in them in detail in Section 4.2.1.8.

4.2 Usability

In this section, we study cases where our algorithm either performs very well or very poorly. We discuss the performance of our algorithm for different styles of programs as well as in different contexts in which to perform term searches.

Generics

Although we managed to make the algorithm work decently with a low amount of generics, extensive use of generics slows it down. Crates in the category "mathematics" are highly generic, and as a result, the average search time in this category is about 15 times longer than the average overall categories (767ms vs 50ms, Table 4). One example is `nalgebra`[1] crate, which

[1]Crates.io, nalgebra library, Accessed: 2024-04-06, URL: https://crates.io/crates/nalgebra

uses generic parameters in almost all of its functions. The slowdown occurs because the wishlist of types grows very large since there are many generic types with different trait bounds.

Tail expressions

We find that tail expressions are one of the best contexts to perform term searches. They are a good fit for both filling holes and also for providing autocompletion suggestions, for the following reasons:

1. Tail expressions usually have a known type. The type is either written explicitly (e.g. a function return type) or can be inferred from context (e.g. all match arms need to have the same type).
2. Once the user starts writing the tail expression, they usually have enough terms available in context to fill the hole. For example, it is common to store `struct` fields in local variables and then combine them into a `struct` only in the tail expression.

Accurate type information is essential for the term search to provide good suggestions. When filling holes, the user has often already put in some extra effort by narrowing down the type of hole. Non-tail expressions, however, often lack enough type information, and thus autocompletion produces no suggestions at all.

Function arguments

We found that letting the algorithm search for parameters of a function call yields good results. This is especially true when the user is working in "exploration mode" and is looking to find different ways of calling the function. Similarly to tail expressions, function calls usually have accurate type information available for the arguments, with some exceptions for generic types. Often, there are also arguments of the right type available in context, so the term search can easily fill them in.

Local variables

In practice, term search is not very useful for generating terms for local variables. Usually, a local variable is bound in a `let`-statement, and it is common to omit its type and let the compiler infer it instead. This, however, means that there is no type information available for the term search. Adding the type explicitly fixes the issue, but this results in non-idiomatic Rust code. In this regard, type inference and term search have opposite goals: One finds types for programs, and the other finds programs for types.

Builder pattern

As discussed in Section 2.5, term search struggles to suggest terms of types using the builder pattern. Suggestions like `Foo::builder().build()` are incomplete but valid suggestions. However, we found that in some cases, such suggestions provide value when the user is writing code in "exploration mode". Such suggestions indicate a way of getting something of the desired type. Now the user has to evaluate if they want to manually call the relevant methods on the builder or if they do not wish to use the builder at all. Without these suggestions, the user may even not know that a builder exists for the type.

Procedural Macros

An interesting observation was that filling holes in the implementation of procedural macros is less useful than usual and can even cause compile errors. The decrease in usability is caused by procedural macros mapping `TokenStream` to `TokenStream` (Rust syntax to Rust syntax), meaning we do not have useful type information available. This is very similar to the builder pattern, so the decrease in usefulness originates from the same reasons. However, procedural macros are

somewhat special in Rust, and they can also raise compile-time errors. For example, one can assert that the input `TokenStream` contains a non-empty `struct` definition. As the term search has no way of knowing that the `TokenStream` has to contain certain tokens, it also suggests other options that validate the rule, causing the error to be thrown.

Formatting
We found that the formatting of the expressions can have a significant impact on the usability of the term search in cases of autocompletion. This is because it is common for LSP clients to filter out suggestions that do not look similar to what the user is typing. Similarity is measured at the level of text with no semantic information available. This means that even though `x.foo()` (method syntax) and `Foo::foo(x)` (universal function call syntax) are the same, the second option is filtered out if the user has typed `x.f` as text wise they do not look similar. This causes some problems for our algorithm, as we decided to use universal function call syntax whenever possible, as this avoids ambiguity. However, users usually prefer method syntax as it is less verbose and easier to understand for humans.

However, this is not a fundamental limitation of the algorithm. One option to solve this would be to produce suggestions using both of the options. That, however, has its issues, as it might overwhelm the user with the number of suggestions if the suggestions are text-wise similar. There can always be options when the user wishes to mix both of the syntaxes, which causes the number of suggestions to increase exponentially as every method call would double the number of suggestions if we would suggest both options.

Foreign function interface crates
We found that for some types of crates, the performance of the term search was significantly worse than for others. It offers a lot more terms per hole by suggesting 303 terms per hole, which is about 15 times more than the average of 20. Such a large number of suggestions is overwhelming to the user, as 300 suggestions do not even fit on the screen. Interestingly, almost all of the terms are found at depth 0, and only a very few are added at later iterations.

A high number of suggestions are caused by those crates using only a few primitive types, mostly integers. For example, in C, it is common to return errors, indexes, and sometimes even pointers as integers. Yet C's application binary interface (ABI) is the only stable ABI Rust has. Foreign function interface (FFI) crates are wrappers around C ABI and therefore often use integer types for many operations.

Searching for an integer, however, is not very useful as many functions in C return integers, which all fit into the hole based on type. For example, the number of terms found per hole reaches 300 already at depth 0, as there are many integer constants that all fit most holes. This means that there is a fundamental limitation of our algorithm when writing C-like code in Rust and working with FFI crates. As the point of FFI crates is to serve as a wrapper around C code so that other crates wouldn't have to, we are not very concerned with the poor performance of term search in FFI crates.

To see how the results for crates with idiomatic Rust code would look, we filtered out all crates from the categories "external-ffi-bindings", "os", and "no-std" (Table 3). We can see that *terms per hole* is the only metric that suffers from C-like code.

Depth	Holes filled	Syntactic matches	Terms per hole	Average time
0	19.0%	2.6%	1.3	0.4ms
1	69.1%	11.1%	3.9	6.5ms
2	75.5%	11.4%	5.8	52.1ms
3	76.7%	11.5%	7.4	84.1ms
4	77.0%	11.4%	8.0	99.1ms
5	77.1%	11.4%	8.2	116.0ms

Table 3: Results without crates from categories *external-ffi-bindings*, *os* and *no-std*. *Holes filled*, *syntactic matches* and *average time* have similar results to overall average. There are about 14 terms less per hole at all depths.

4.3 Limitations of the methods

In this section, we highlight some limitations of our evaluation. We point out that "holes filled" is a too permissive metric, and "syntactic matches" is too strict. Ideally, we want something in between, but we don't have a way to measure it.

Resynthesis

Metric "holes filled" does not reflect the usability of the tool very well. This would be a useful metric if we used it as a proof search. When searching for proofs, we often care that the proposition can be proved rather than which of the possible proofs it generates. This is not the case for regular programs with side effects. For them, we only care about semantically correct terms, e.g. do what the program is supposed to do. Other terms can be considered noise, as they are programs that no one asked for.

Syntactic matches (equality) is a too strict metric as we care about the semantic equality of programs. The metric may depend more on the style of the program and the formatting than on the real capabilities of the tool. Syntactic matches also suffer from squashing multiple terms to the `Many` option, as the new holes produced by *Many* are not what was written by the user.

Average time and number of terms per hole are significantly affected by a few categories that some may consider outliers. We have decided not to filter them out to also show that our tool is a poor fit for some types of programs.

Average execution can also be criticized for being irrelevant. Having the IDE freeze for a second once in a while is not acceptable, even if at other times the algorithm is very fast. To also consider the worst-case performance, we have decided to also measure latency. However, we must note that we only measure the latency of our algorithm. While using the tool in the IDE, the latency is higher due to LSP communication and the IDE also taking some time. We only measure the latency of our algorithm, as other sources of latency are outside of our control and highly dependent on the environment.

Usability

This section is based on the personal experience of the author and may therefore not reflect the average user very well. Modeling the average user is a hard task on its own and would require us to conduct a study on it. As studying the usage of IDE tools is outside the scope of this thesis, we only attempt to give a general overview of the strengths and weaknesses of the tool. Different issues may arise when using the tool in different contexts.

5 Future work

In this section, we will discuss some of the work that could be done to improve term search in `rust-analyzer`. Some of these topics consist of features that were not in the scope of this thesis. Others focus on improving the `rust-analyzer` functionality overall.

More permissive borrow checking

The current borrow checking algorithm we implemented for `rust-analyzer` is rather conservative and also forbids many of the correct programs. This decreases the usefulness of term search whenever reference types are involved. The goal would be to make the borrow checking algorithm in `rust-analyzer` use parts of the algorithm that are in the official compiler but somehow allow borrow checking also on incomplete programs. Lowering incomplete programs (the user is still typing) to MIR and performing borrow checking incrementally is a complex problem, however, we believe that many other parts of the `rust-analyzer` could benefit from it.

Smarter handling of generics

In projects with hundreds of functions that take generic parameters our algorithm effectiveness decreases. One of the main reasons for this is that we fully normalize all types before working with them. In the case of types and functions that have generic parameters, this means substituting the generic parameters. However, that is always not required. Some methods on types with generic parameters do not require knowing exact generic parameters and therefore can be used without substituting in the generic types. Some examples of it are `Option::is_some` and `Option::is_none`. Others only use some of the generic parameters of the type. If not all generic parameters are used, we could avoid substituting the generic types that are not needed, as long as we know that we have some options available for them.

Development of more tactics

A fairly obvious improvement that we believe still should be touched on is the addition of new tactics. The addition of new tactics would allow usage in a new context. Some ideas for new tactics:

- Tuple projection - very similar to struct projection. Very easy to add.
- Macro call - similarly to function calls, macros can be used to produce terms of unexplored types. As macros allow more custom syntax and work at the level of metaprogramming, adding them can be more complex.
- Higher-order functions - generating terms for function types is more complex than working with simple types. On the other hand, higher-order functions would allow the usage of term search in iterators and therefore increase its usefulness by a considerable margin.
- Case analysis - Perform a case split and find a term of suitable type for all the match arms. May require changing the algorithm slightly as each of the match arms has a different context.

Machine learning based techniques

We find that machine-learning-based techniques could be used to prioritize generated suggestions. All the terms would still be generated by term search and would be valid programs by construction, which is a guarantee that LLMs cannot have. On the other hand, ordering suggestions is very hard to do analytically, and therefore we believe that it makes sense to train a model for it. With a better ordering of suggestions, we can be more permissive and allow suggestions that do not affect the type of the term (endofunctions). For example, suggestions for builder patterns could be made more useful by also suggesting some setter methods.

LSP response streaming

Adding LSP response streaming is an addition to `rust-analyzer` that would also benefit term search. Response streaming would be especially helpful in the context of autocompletion. It would allow us to incrementally present the user autocompletion suggestions, meaning that latency would become less of an issue. With the latency issue solved, we believe that term-search-based autocompletion suggestions could be turned on by default. Currently, the main reason for making them opt-in was that the autocompletion is already slow in `rust-analyzer` and term search makes it even slower.

6 Conclusion

In this thesis, our main objective was to implement a term search for the Rust programming language. We achieved it by implementing it as an addition to `rust-analyzer`, the official LSP server for Rust.

First, we gave an overview of the Rust programming language to understand the context of our work. We are focusing on the type system and the borrow checking, as they are two fundamental concepts in Rust.

After that, we gave an overview of term search and the tools for it. We focused on the tools used in Agda, Idris, Haskell, and StandardML. We analyzed both their functionality and the algorithms they use. By comparing them to one another, we laid the groundwork for our implementation.

After that, we covered the LSP protocol and some of the autocompletion tools to gain some understanding of the constraints we have when trying to use the term search for autocompletion.

The term search algorithm we implemented is based on the tools used in Agda, Idris, Haskell, and StandardML. We took a different approach from the previous implementations by using a bidirectional search. The bidirectional approach allowed us to implement each tactic in the direction that was the most natural fit for it. This resulted in a rather simple implementation of tactics that achieved relatively high performance.

To evaluate the performance of the algorithm, we ran it on existing open-source projects. For measuring the performance, we chose the top 5 projects from the most popular categories on crates.io, the Rust community's crate registry. This resulted in 155 crates.

We added term-search-based autocompletion suggestions to evaluate the usability of term search for autocompletion. With its small depth, the algorithm proved to be fast enough and resulted in more advanced autocompletion suggestions compared to the usual ones. As the autocompletion in `rust-analyzer` is already rather slow, the feature is disabled by default, yet all the users of it can opt into it.

6 Bibliography

[1] S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How Programmers Interact with Code-Generating Models," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023, doi: 10.1145/3586030 .

[2] G. Li, "An Affine Type System with Hindley-Milner Style Type Inference," p. , 2022.

[3] G. Nadathur, "A Proof Procedure for the Logic of Hereditary Harrop Formulas," Duke University, USA, 1992.

[4] P. Wadler, "Propositions as Types," *Commun. ACM*, vol. 58, no. 12, pp. 75–84, Nov. 2015, doi: 10.1145/2699407 .

[5] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda – A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78. [Online]. Available: https://doi.org/10.1007/978-3-642-03359-9_6

[6] F. Lindblad and M. Benke, "A Tool for Automated Theorem Proving in Agda," in *Types for Proofs and Programs*, J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 154–169.

[7] L. Skystedt, "A New Synthesis Tool for Agda," 2022. [Online]. Available: https://hdl.handle.net/20.500.12380/305712

[8] C. Angiuli, E. Cavallo, K.-B. Hou (Favonia), R. Harper, and J. Sterling, "The RedPRL Proof Assistant (Invited Paper)," in *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Oxford, UK, 7th July 2018*, F. Blanqui and G. Reis, Eds., in Electronic Proceedings in Theoretical Computer Science, vol. 274. Open Publishing Association, 2018, pp. 1–10. doi: 10.4204/EPTCS.274.1 .

[9] J. Sterling and R. Harper, "Algebraic Foundations of Proof Refinement," *CoRR*, 2017, [Online]. Available: http://arxiv.org/abs/1703.05215

[10] N. G. de Bruijn, "Telescopic mappings in typed lambda calculus," *Information and Computation*, vol. 91, no. 2, pp. 189–204, 1991.

[11] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 5, pp. 552–593, 2013, doi: 10.1017/S095679681300018X .

[12] J. Lubin, N. Collins, C. Omar, and R. Chugh, "Program sketching with live bidirectional evaluation," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020, doi: 10.1145/3408991 .

[13] J. Fiala, S. Itzhaky, P. Müller, N. Polikarpova, and I. Sergey, "Leveraging Rust Types for Program Synthesis," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023, doi: 10.1145/3591278 .

[14] N. Polikarpova and I. Sergey, "Structuring the synthesis of heap-manipulating programs," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019, doi: 10.1145/3290385 .

[15] S. D. Bergmann, *Compiler Design: Theory, Tools, and Examples*, 1st ed. McGraw-Hill Professional, 1994. doi: 10.31986/issn.2689-0690_rdw.oer.1001 .

[16] D. Barros, S. Peldszus, W. K. G. Assunção, and T. Berger, "Editing support for software languages: implementation practices in language server protocols," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, in MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 232–243. doi: 10.1145/3550355.3552452 .

[17] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code Prediction by Feeding Trees to Transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 150–162. doi: 10.1109/ICSE43902.2021.00026 .

[18] Q. Zheng *et al.*, "CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, in KDD '23., Long Beach, CA, USA, : Association for Computing Machinery, 2023, pp. 5673–5684. doi: 10.1145/3580305.3599790 .

[19] W. Crichton, "The Usability of Ownership," *arXiv e-prints*, p. arXiv:2011.06171, Nov. 2020, doi: 10.48550/arXiv.2011.06171 .

[20] D. G. Corneil and R. M. Krueger, "A Unified View of Graph Searching," *SIAM J. Discret. Math.*, vol. 22, no. 4, pp. 1259–1276, Jul. 2008, doi: 10.1137/050623498 .

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. in Addison-Wesley professional computing series. Addison-Wesley, 1995. [Online]. Available: https://books.google.ee/books?id=tmNNfSkfTlcC

[22] R. Péter, "H. G. Rice. Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society, vol. 74 (1953) pp. 358–366.," *Journal of Symbolic Logic*, vol. 19, no. 2, pp. 121–122, 1954, doi: 10.2307/2268870 .

[23] J. Nielsen, *Usability engineering*. Morgan Kaufmann, 1994.

[24] R. C. Thomas, A. Karahasanovic, and G. E. Kennedy, "An investigation into keystroke latency metrics as an indicator of programming performance," in *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, in ACE '05., Newcastle, New South Wales, Australia, : Australian Computer Society, Inc., 2005, pp. 127–134.

Appendix 1: Non-Exclusive License for Reproduction and Publication of a Graduation Thesis

I, Tavo Annus

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Term Search in Rust", supervised by Philipp Joram
   1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

12.05.2024

Appendix 2: List of crates

Category	Crate
algorithms	rand_core-0.6.4
algorithms	rand-0.8.5
algorithms	rand_chacha-0.3.1
algorithms	num-traits-0.2.18
algorithms	crossbeam-utils-0.8.19
api-bindings	socket2-0.5.6
api-bindings	flate2-1.0.28
api-bindings	zstd-safe-7.1.0
api-bindings	openssl-0.10.64
api-bindings	libloading-0.8.3
asynchronous	mio-0.8.11
asynchronous	tokio-1.37.0
asynchronous	futures-0.3.30
asynchronous	tokio-util-0.7.10
asynchronous	tracing-0.1.40
command-line-interface	textwrap-0.16.1
command-line-interface	clap_lex-0.7.0
command-line-interface	clap-4.5.4
command-line-interface	clap_derive-4.5.4
command-line-interface	os_str_bytes-7.0.0
command-line-utilities	crossterm-0.27.0
command-line-utilities	inferno-0.11.19
command-line-utilities	names-0.14.0
command-line-utilities	honggfuzz-0.5.55
command-line-utilities	self_update-0.39.0
concurrency	parking_lot_core-0.9.9
concurrency	crossbeam-channel-0.5.12
concurrency	parking_lot-0.12.1
concurrency	lock_api-0.4.11
concurrency	crossbeam-utils-0.8.19
cryptography	block-buffer-0.10.4
cryptography	ppv-lite86-0.2.17
cryptography	rustls-0.23.4
cryptography	digest-0.10.7
cryptography	sha2-0.10.8
data-structures	smallvec-1.13.2
data-structures	hashbrown-0.14.3

Category	Crate
data-structures	indexmap-2.2.6
data-structures	semver-1.0.22
data-structures	bytes-1.6.0
database	rusqlite-0.31.0
database	postgres-types-0.2.6
database	diesel-2.1.5
database	rocksdb-0.22.0
database	librocksdb-sys-6.20.3
development-tools	proc-macro2-1.0.79
development-tools	quote-1.0.35
development-tools	log-0.4.21
development-tools	syn-2.0.57
development-tools	autocfg-1.2.0
embedded	bitvec-1.0.1
embedded	ciborium-io-0.2.2
embedded	ciborium-0.2.2
embedded	oorandom-11.1.3
embedded	ciborium-ll-0.2.2
encoding	serde-1.0.197
encoding	byteorder-1.5.0
encoding	url-2.5.0
encoding	base64-0.22.0
encoding	serde_json-1.0.115
external-ffi-bindings	libc-0.2.153
external-ffi-bindings	winapi-util-0.1.6
external-ffi-bindings	winapi-0.3.9
external-ffi-bindings	linux-raw-sys-0.6.4
external-ffi-bindings	openssl-sys-0.9.102
filesystem	rustix-0.38.32
filesystem	walkdir-2.5.0
filesystem	glob-0.3.1
filesystem	remove_dir_all-0.8.2
filesystem	which-6.0.1
game-development	egui-0.27.1
game-development	eframe-0.27.1
game-development	gpu-alloc-0.6.0
game-development	gpu-alloc-types-0.3.0
game-development	egui-winit-0.27.1

Category	Crate
graphics	exr-1.72.0
graphics	rgb-0.8.37
graphics	gpu-alloc-0.6.0
graphics	gpu-alloc-types-0.3.0
graphics	tiny-skia-path-0.11.4
gui	stdweb-0.4.20
gui	stdweb-internal-macros-0.2.9
gui	winit-0.29.15
gui	stdweb-derive-0.5.3
gui	stdweb-internal-runtime-0.1.5
hardware-support	num_cpus-1.16.0
hardware-support	cpufeatures-0.2.12
hardware-support	portable-atomic-1.6.0
hardware-support	num_threads-0.1.7
hardware-support	safe_arch-0.7.1
mathematics	rust_decimal-1.35.0
mathematics	nalgebra-0.32.5
mathematics	smawk-0.3.2
mathematics	crypto-bigint-0.5.5
mathematics	bigdecimal-0.4.3
multimedia	tiff-0.9.1
multimedia	png-0.17.13
multimedia	image-0.25.1
multimedia	rgb-0.8.37
multimedia	exr-1.72.0
network-programming	bytes-1.6.0
network-programming	socket2-0.5.6
network-programming	tokio-1.37.0
network-programming	hyper-1.2.0
network-programming	h2-0.4.3
no-std	libc-0.2.153
no-std	bitflags-2.5.0
no-std	rand_core-0.6.4
no-std	rand-0.8.5
no-std	serde-1.0.197
os	libc-0.2.153
os	getrandom-0.2.12
os	nix-0.28.0

Category	Crate
os	rustix-0.38.32
os	winapi-0.3.9
parser-implementations	syn-2.0.57
parser-implementations	url-2.5.0
parser-implementations	toml-0.8.12
parser-implementations	serde_json-1.0.115
parser-implementations	time-0.3.34
parsing	byteorder-1.5.0
parsing	toml-0.8.12
parsing	nom-7.1.3
parsing	minimal-lexical-0.2.1
parsing	semver-parser-0.10.2
rendering	unic-ucd-segment-0.9.0
rendering	unic-segment-0.9.0
rendering	gl_generator-0.14.0
rendering	khronos_api-3.1.0
rendering	unic-ucd-version-0.9.0
rust-patterns	once_cell-1.19.0
rust-patterns	scopeguard-1.2.0
rust-patterns	itertools-0.12.1
rust-patterns	lazy_static-1.4.0
rust-patterns	thiserror-1.0.58
science	num-traits-0.2.18
science	num-complex-0.4.5
science	num-integer-0.1.46
science	num-bigint-0.4.4
science	num-rational-0.4.1
text-processing	regex-1.10.4
text-processing	aho-corasick-1.1.3
text-processing	strsim-0.11.0
text-processing	unicode-bidi-0.3.15
text-processing	regex-automata-0.4.6
wasm	reqwest-0.12.2
wasm	wasm-bindgen-0.2.92
wasm	js-sys-0.3.69
wasm	uuid-1.8.0
wasm	wasi-0.13.0+wasi-0.2.0
web-programming	hyper-1.2.0

Category	**Crate**
web-programming	http-1.1.0
web-programming	h2-0.4.3
web-programming	httparse-1.8.0
web-programming	url-2.5.0

Appendix 3: Per category results

Category	Holes filled	Syntactic matches	Suggestions per expr	Avg time
algorithms	72.7%	8.6%	4.0	7.8ms
api-bindings	66.9%	8.2%	4.6	10.6ms
asynchronous	69.2%	10.6%	3.0	4.6ms
command-line-interface	69.4%	5.3%	2.3	7.6ms
command-line-utilities	79.7%	17.2%	3.5	14.2ms
concurrency	74.6%	17.1%	5.9	4.2ms
cryptography	76.3%	10.5%	4.0	23.4ms
data-structures	58.4%	5.0%	2.7	5.2ms
database	58.4%	18.9%	28.3	47.6ms
development-tools	80.2%	17.2%	4.0	14.4ms
embedded	60.9%	9.8%	3.4	21.8ms
encoding	67.9%	10.1%	3.6	27.6ms
external-ffi-bindings	84.2%	11.5%	303.1	48.2ms
filesystem	69.6%	8.8%	2.8	7.0ms
game-development	68.8%	12.3%	4.9	66.8ms
graphics	80.8%	15.3%	4.4	17.0ms
gui	83.2%	9.4%	3.6	36.4ms
hardware-support	99.2%	0.9%	15.8	36.8ms
mathematics	73.1%	7.5%	7.1	767.0ms
multimedia	82.1%	15.1%	4.9	59.0ms
network-programming	78.8%	11.3%	3.3	9.8ms
no-std	52.9%	8.6%	76.5	14.0ms
os	62.3%	6.9%	77.7	11.2ms
parser-implementations	81.7%	13.2%	5.4	30.8ms
parsing	68.2%	5.4%	4.6	25.4ms
rendering	76.0%	21.1%	5.1	2.2ms
rust-patterns	60.4%	9.2%	2.8	6.8ms
science	90.2%	8.0%	5.8	81.8ms
text-processing	82.4%	14.0%	6.4	10.4ms
wasm	80.5%	16.6%	13.0	103.0ms
web-programming	77.5%	11.6%	4.0	10.6ms

Table 4: Per category results with depth = 2

Appendix 4: Reduced list of crates

Category	Crate
algorithms	rand-0.8.5
api-bindings	socket2-0.5.6
asynchronous	mio-0.8.11
command-line-interface	clap-4.5.4
command-line-utilities	crossterm-0.27.0
concurrency	parking_lot_core-0.9.9
cryptography	digest-0.10.7
data-structures	hashbrown-0.14.3
database	rusqlite-0.31.0
development-tools	syn-2.0.58
embedded	bitvec-1.0.1
encoding	serde-1.0.197
external-ffi-bindings	libc-0.2.153
filesystem	rustix-0.38.32
game-development	gpu-alloc-0.6.0
graphics	rgb-0.8.37
gui	stdweb-0.4.20
hardware-support	num_cpus-1.16.0
mathematics	crypto-bigint-0.5.5
multimedia	png-0.17.13
network-programming	bytes-1.6.0
no-std	libc-0.2.153
os	libc-0.2.153
parser-implementations	syn-2.0.58
parsing	byteorder-1.5.0
rendering	unic-ucd-version-0.9.0
rust-patterns	once_cell-1.19.0
science	num-traits-0.2.18
text-processing	regex-1.10.4
wasm	uuid-1.8.0
web-programming	url-2.5.0