型に基づくRust API探索の手法と実装

Methods and Implementation of Type-based Rust API Search

後藤将之・松下祐介・五十嵐淳（京都大学）

本研究の貢献

・型に基づくAPI探索：関数型をクエリとして受け取り、クエリを満たすAPI関数を探索
→ 例：クエリが fn(i32) -> i32 だと **abs** : fn(i32) -> i32 や **id** : ∀T. fn(T) -> T は結果の一つ

既存研究・ツール

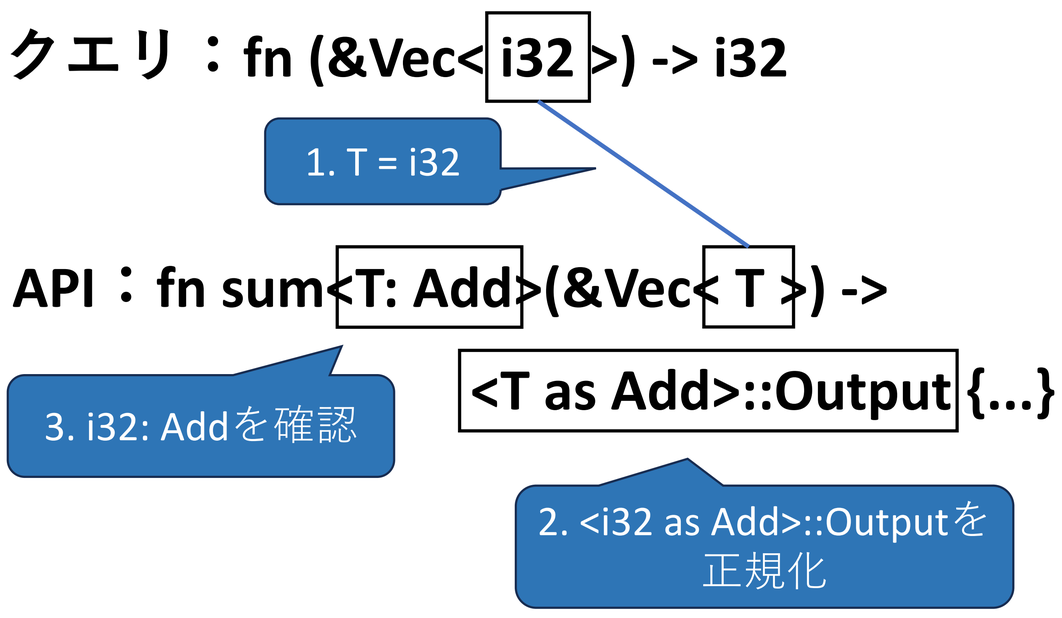
1. **Rust : Roogle** (https://roogle.hkmatsumoto.com)
→ 単独のAPIのみ探索・未対応のRustの機能がある

2. **Haskell：Hoogle+** [James et al. 2020]
→ `map head (group “aab”)`のような合成された関数の探索が可能

本研究では以下の手法の提案と実装を行う

1. Roogleでは困難な関連型とトレイト制約をサポートする単独API探索
2. Hoogle+に着想を得た、厳しい制約のもとでのメソッドチェインの探索

貢献１：関連型とトレイト制約を含む単独API関数の探索



▶ 左の探索の流れ

1. 関連型は型変数が定まらなければ確定しないので、一度関連型を無視して型変数を決定する
左の例では **T = i32**

2. 1で求めた型変数のもとで関連型を正規化し、クエリとの一致を確認する
左の例では **<i32 as Add>::Output** は **i32** となり、クエリと一致

3. 1で求めた型変数のもとでトレイト制約を確認する
左の例では解くべき制約は **i32: Add** となり、これは成り立つ

貢献２：メソッドチェインの探索

▶ 例：クエリが fn (&String) -> Vec< char > ➡ .as_ref() .chars() .collect() は結果の一つ

as_ref : fn(&String) -> &str ， chars : fn(&str) -> Chars ， collect : fn(Chars) -> Vec< char >

▶ 探索の流れ

1. &String型の値を引数に取るメソッドを探索 → as_ref は結果の一つ
2. Vec< char >型の値を返すメソッドとして collect を探索
3. クエリをfn (&str) -> Chars としてメソッドを探索 → chars は結果の一つ

▶ 探索空間の爆発を防ぐため、探索に
メソッド数は3以下 ・ 引数はselfのみ・最後のメソッドはcollect、という制約を課す

・キャッシュ
キャッシュを使用して複数回同じクエリで探索することを防ぎ、探索時間を短縮させる

・フィルタリング
無意味な型変換メソッドfrom, intoを含むメソッドチェインを除き、不要な結果を減少させる

実験結果１：単独API関数の探索

クエリ：fn (&i32, &i32) -> i32
ライブラリ：std, core, alloc, proc_macro
環境：MacBook Pro 2020 (intel core i5, 4コア, 16GB)

結果の上位3つ

API関数	型
add	fn (&i32, &i32) -> <i32 as Add>::Output
bitand	fn (&i32, &i32) -> <i32 as BitAnd>::Output
bitor	fn (&i32, &i32) -> <i32 as BitOr>::Output

実行時間：0.060s

実験結果２：メソッドチェインの探索

クエリ：fn (&str) -> Vec< char >
ライブラリ：std, core, alloc, proc_macro
環境：MacBook Pro 2020 (intel core i5, 4コア, 16GB)

結果の一部

メソッド2つの結果全て	メソッド3つの結果の上位4つ
.chars() .collect()	.as_mut() .chars() .collect()
.escape_debug() .collect()	.as_mut() .escape_debug() .collect()
.escape_default() .collect()	.as_mut() .escape_default() .collect()
.escape_unicode() .collect()	.as_mut() .escape_unicode() .collect()

最適化手法の有無による比較

キャッシュ	フィルタリング	実行時間 (秒)	結果の個数
なし	なし	151.7641	16169
あり	なし	22.6125	16169
なし	あり	30.6004	3292
あり	あり	22.1647	3292

→ 最適化手法により、時間が約**85%**、結果の個数が約**80%**減少

今後の課題

1. アルゴリズムの形式化と正当性の証明
2. 探索における制約の緩和