arXiv:2602.18534v1 [cs.SE] 20 Feb 2026

针对使用外部库的项目的经验证代码转换

Hanliang Zhang[1,2], Arindam Sharma[2], CristinaDavid[2], MengWang[2], Brandon Paulsen[1], Daniel Kroening[1], Wenjia Ye[2], 和 Taro Sekiyama[3]

[1] 亚马逊公司，英国 {hlz,bpaulse,dkr}@amazon.co.uk[2] 布里斯托大学，英国
{arindam.sharma,cristina.david,meng.wang,wenjia.ye}@bristol.ac.uk[3] 日本国立信息学研究所 sekiyama@nii.ac.jp

摘要。 大语言模型（LLM）在程序翻译方面展现出良好前景，尤其适用于将系统代码迁移至 Rust 等内存安全语言。然而，当源程序依赖外部库时，现有方法往往难以应对：大语言模型经常臆造不存在的目标 API，且无法生成使调用可用的导入项；此外，当代码操作不透明的库定义类型时，验证语义等价性也颇具挑战。我们提出了一种将具有外部依赖项的 *Go* 项目转换为 *Rust* 的翻译与验证框架。我们的方法结合了（i）将 Go 库 API 映射到 Rust API 的检索机制，以及（ii）一种跨语言验证流水线：在验证输入输出等价性之前，仅使用公共库 API 合成适配器，从而在存在不透明库类型的情况下建立语言互操作性。我们在六个具有复杂外部依赖项的真实 Go 代码仓库上评估了系统。通过支持对操作不透明库定义类型的代码进行经验证的翻译，我们的方法显著提高了编译成功率和等价性成功率（依赖最复杂的案例中最高达到 100%；平均约提高 2 倍）。

关键词： 代码转换 · Rust · Go · 大语言模型。

1 引言

程序翻译是一项具有众多应用的重要任务，已受到学术界 [52, 55, 57], 工业界 [19, 28, 48],和政府机构 [2, 13]的广泛关注。该领域的早期工作是在编译器 [36, 45], 的背景下开展的，但近年来，源到源翻译逐渐受到欢迎。沿着这一方向，许多方法开发了基于规则的系统 [1,3, 6, 11, 17, 21, 26, 29, 30, 34, 35, 47, 50, 54, 61, 63, 65], ，但这些翻译通常不符合目标语言的习惯，尤其是对于 Rust [19, 42]这类目标语言而言。较新的方法将大语言模型结合到语义验证系统中；大语言模型能够生成符合语言习惯但未经验证的翻译，而语义验证系统则包括差分

测试 [4, 9, 28, 32, 33, 51, 62, 66], 模糊测试 [18, 33], 或形式化验证 [10, 60],, 以确保正确性。当前最先进的方法会一次性翻译整个代码库 [28, 52, 55, 62]。

尽管取得了这些进展，鉴于两个关键限制，先前的程序翻译工作一直避免翻译大量使用外部库的代码库:

首先，当源代码使用外部库时，大语言模型在代码转换方面的性能会下降 [24]。大语言模型会臆造目标语言中不存在的库和库 API，并且会误用现有的库 API。

其次，当源函数及其翻译使用外部库时，先前的技术通常无法推理二者的语义等价性。许多技术根本无法推理函数之间的等价性 [12, 14, 27, 33,38,42,52,55–57, 59, 64] ——它们依赖端到端测试（例如执行类似命令行界面的编译后二进制文件）或手写单元测试来验证翻译的正确性。那些确实 尝试推理函数等价性的技术所作的假设，在源函数及其翻译使用外部库时往往并不成立。大多数技术都假设源函数中的输入参数类型与目标语言中的输入参数类型在结构上完全相同 [4, 10, 18, 28, 32, 51, 60, 62, 66]。当这些输入类型在外部库中定义时，该假设往往不成立。此外，所有这些工作都假设可以直接访问复杂数据类型的所有字段；而在支持不透明类型的现代语言中，这是不可能做到的，或者极其困难 [66]。

贡献。 据我们所知，本文提出了首个能够同时解决这两项局限的代码转换方法。我们的方法能够验证具有结构上不相似且不透明的输入参数类型的函数对之间的等价性，并采用检索增强生成架构，大幅减少外部库 API 的幻觉。鉴于其实际相关性 [18] 以及 Go 具有明确定义的库概念，我们重点介绍将完整的 Go 代码仓库转换为 Rust；但我们的思路适用于大多数现代语言。我们在六个具有非平凡依赖项的真实 Go 代码仓库上评估了该方法，同时考察可编译性和差分输入输出等价性。在所有基准测试中，这两项指标均取得了平均 95.83% 的成功率。在依赖项最多的情况下，成功率最高提升了 100%，平均提升超过 2 倍。

2 动机示例

我们通过图 1 中 age 代码仓库的一个示例来说明关键挑战 [20] ，其中展示了一个执行密码学操作的 Go 函数及其 Rust 翻译版本。我们的方法依赖大语言模型作为主要的代码生成器。

挑战 ***1***：跨语言库与 ***API*** 映射。在图 1 中，在 Go 端，函数调用 `sha512.New()` 和 `h.Write(pk.Seed())`（Go 第 5–6 行）。虽然意图很明确，但翻译器仍必须决定哪一个

```
导入 "crypto/sha512"
导入 "crypto/ed25519"
函数 ed25519PrivateKeyToCurve25519(
        pk ed25519.PrivateKey) []字节 {
    h := sha512.New()
    h.Write(pk.Seed())
    out := h.Sum(nil)
    return out [:Curve25519.标量大小]
}
```

```
use sha2::{SHA-512 , Digest};
use ed25519_dalek::SigningKey;
pub(crate) fn ed25519 私钥转 Curve25519
        (sk: &SigningKey) -> [u8; 32] {
    let mut 哈希器 =Sha512::new();
    哈希器.update(sk.to_bytes());
    let digest = hasher.finalize();
    let mut output = [0u8; 32];
    output.copy_from_slice(&digest [..32]);
    output
}
```

图 1：来自 age [20] 的 Go 函数及其 Rust 翻译

```
导入 "crypto/rsa"
导入 "golang.org/x/crypto/ssh"

类型 RSAIdentity 结构体 {
        k *rsa.PrivateKey
        sshKey ssh.PublicKey
}
```

图 2：Go 类型 RSAIdentity

```
use rsa::RsaPrivateKey; use
russh::keys::PublicKey; pub struct
RsaIdentity { pub(crate) k: RsaPrivateKey ,
pub(crate) ssh_key: PublicKey, }
```

图 3：已翻译的Rust 类型 RsaIdentity

目标 crate 和 API 与源调用最匹配。具体而言，一种可能的 Rust 翻译会使用 `Sha512::new()` 构造哈希器（Rust 第 5 行），通过 `sk.to_bytes()` 传入由签名密钥派生的字节（Rust 第 6 行），并最终计算摘要（Rust 第 7–9 行）。依赖大语言模型的参数化知识时，这一步很脆弱：模型可能会选择不正确的或不存在的目标 crate/API，导致编译失败（见第 6 节）。

挑战 *1.1*：*Rust* 导入解析。在 Rust 中，选对目标 API 只是成功的一半：翻译还必须找出使调用可用的导入，才能在调用位置使用该 API。我们将此视为挑战 1 的一项关键子挑战。这很脆弱，因为 Rust 可调用性往往是非局部的。首先，许多方法由特征提供，只有在定义该特征的模块位于作用域内时才可用。其次，crate 通常会通过再导出重组其公共接口，因此可导入路径往往不同于定义所在的模块。结果是，大语言模型经常生成引用了正确 API、却因导入项不正确而无法编译的代码。

图 1 展示了这两种陷阱。调用 `Sha512::new()` 需要导入 `Digest` 特征（重新导出为 `sha2::Digest`），而 `SigningKey` 定义在私有模块中，但在 crate 根处公开。对于这个示例，翻译器因此必须还原正确的导入项`sha2::{Sha512, Digest}` 和 `ed25519_-dalek::SigningKey` （Rust 第 1–2 行）。

挑战 ***2***：使用不透明库类型进行跨语言验证。 即使翻译器正确解析了 crate 和 API（挑战 1），并生成了可调用的 Rust 代码且导入项正确（挑战 1.1），翻译结果仍然缺乏等价性证据。验证仓库级翻译需要进行跨语言比较，但 Go 和 Rust 既没有共用的运行时，也没有共用的值表示。

```
type 私钥 struct {
    公钥                    // 公开部分。
    D          *big.Int  // 私钥指数
    素数       []*big.Int // N 的素因子，
                          // 有 >= 2 个元素。

    // 预计算包含能够
    // 加速 RSA 运算（如果可用）。它
    // 不得被修改。
    预计算 PrecomputedValues
}
```

图 4：rsa.PrivateKey 的定义。

```
…
- 库 API 的翻译
建议使用以下内容
来自 crate 的 API crate
1. ${api_1.import_paths}$: ${api_-
1.签名}$，${api _1.文档}$
2. ${api _2.导入 _路径}$: ${api _-
2.签名}$，${api _2.docs}$
…
```

图5：翻译提示中的 RAG增强上下文。

原则上，我们希望至少 通过在单元测试输入上执行 Go 源代码，然后检查 Rust 翻译是否产生匹配的输出，来验证翻译后的函数。然而，这种看似直接的方法在实践中行不通，因为 Go 和 Rust 之间无法直接交换值，而且相关输入/输出通常涉及库定义类型（例如 `ed25519.PrivateKey` 和`SigningKey`）。此类类型通常是不透明的 （具有私有字段），缺乏稳定的（反）序列化接口，而且只能通过精心挑选的公共 API 来构造或检查。因此，验证不能依赖字段级转换或结构匹配；它需要仅使用公共库 API 合成保持行为的适配器 。

对于用户定义的包装类型，例如 `RSAIdentity` （图 2）及其 Rust 翻译版本 `RsaIdentity` （图 3），这种困难尤为突出。它们封装了来自不同生态系统的密码学密钥类型（例如 `rsa.PrivateKey`（图4）与 `rsa::RsaPrivateKey`）。在这种情况下，即使只是构造一个对应的测试输入，也需要处理接口类型、特征作用域内的方法以及特定于crate的编码，因此跨语言差分测试极具挑战性。具体而言，在 `rsa.PrivateKey`、`ssh.PublicKey` 与其 Rust 对应类型之间合成转换时，不能依赖按字段进行的结构化转换：`ssh.PublicKey` 是一个接口，而 `rsa.PrivateKey`包含嵌套库类型（例如 `big.Int`和 `PrecomputedValues`，图4），这些类型的值必须满足特定于库的不变量。因此，要实现正确的互操作，通常需要使用库的公共构造函数和（反）序列化例程，而不是直接访问或重建内部字段。

3 高级方法

从较高层面来看，我们的流水线执行模块化的函数级翻译：每个类型定义和函数均由大语言模型单独翻译，并以一份紧凑的依赖摘要 为指导，该摘要捕获范围内的类型以及直接被调用函数的签名。这一设计延续了先前基于大语言模型的模块化翻译方法的思路，例如**Oxidizer** [62]。

3.1 阶段 1：跨语言库映射（挑战 1）

如第2节所述，大语言模型往往无法确定 哪个 RUSTcrate 和 API 最匹配给定的 GO 调用。即便对于标准库用法，正确的对应项也可能位于第三方 Rust crate 中，而非 `std`，并且多个候选 crate 可能提供相互重叠的功能。

为缓解这一问题，我们在每个翻译提示中加入明确的库解析上下文。具体而言，对于每个 Go 函数，我们会 (i) 提取其调用的 Go 库 API 集合，并 (ii) 检索一小组语义相关的候选 *Rust API*。这些候选项通过下文简要介绍的检索增强流水线获取（第 4 节将详述）：

Crate 匹配。 首先，我们将每个 Go 包 `pkg`映射到对应的 Rust crate `crate`。我们假设映射是单射： 每个 Go 包都映射到单个 Rust crate，且 `pkg`中的所有 API 都在 `crate`内解析。这样可以使检索更有针对性，并控制提示的预算。虽然可以放宽这一限制，允许每个包对应多个 crate，但我们发现，单射是实用且有效的近似假设。

我们通过结合 (i) 大语言模型根据包级文档摘要提出的匹配结果，以及 (ii) 在`crates.io`上进行的基于关键词的搜索来获取候选 crate，随后根据文档进行重排序，并在主翻译循环开始前预先计算映射。以图 1 中的示例为例，这会为 **crypto/sha512** 映射到 sha2，为 **crypto/ed2551925519_映射到 ed**dalek。

按 *crate* 构建知识库。对于每个匹配的目标 crate`crate`，我们都会预先构建一个该 crate 专属的知识库 $\mathcal{K}_{\texttt{crate}}$，其中包含其公共 API。对于源函数中使用的每个库 API，我们都会通过文本描述查询 $\mathcal{K}_{\texttt{crate}}$，该描述将 Go API 名称及其文档结合起来。例如，对于 `sha512.New`，我们构造如下查询：Query`sha512.New` = `sha512.New`：“*New* 返回一个新的 *hash.Hash*，用于计算 *SHA-512* 校验和。”，并从$\mathcal{K}_{\texttt{sha2}}$中检索候选项。在我们的系统中，`Sha512::new`排名最高。

Rust 导入解析。鉴于第 2 节所述，Rust 中的导入解析并非易事，我们扩展了 crate 专属知识库 $\mathcal{K}_{\texttt{crate}}$，使其不仅记录有哪些API，还记录如何导入和使用这些 API。具体而言， $\mathcal{K}_{\texttt{crate}}$ 中的每个条目都包含一个可公开使用的 API、其文档，以及一个或多个有效的公共导入路径。在我们的示例中，`sha512.New`对应的最高匹配项是`Sha512::new`，并附带使调用可用的导入项`sha2::{Sha512, Digest}`，从而支持图 1 中的 Rust 翻译。同样，Go 调用`pk.Seed()` (Go 第 6 行)通过 $\mathcal{K}_{\texttt{ed25519 dalek}}$解析为`SigningKey::to_bytes()` (Rust 第 6 行)，使用正确的根级导入路径`ed25519_-¯dalek::SigningKey`。

3.2 阶段2：生成翻译

如前所述，借助库知识库，我们扩展了标准依赖摘要——该摘要涵盖范围内类型以及

```
消息 RsaIdentity { optional 字节
k = 1; 字节 SSH 密钥 = 2; }
```

图 6：PROTOBUF 类型（T_{Proto}）的模式，适用于类型 `RSAIdentity`（图 2）和 `RsaIdentity`（图 3）

```
类型 ProtoRSAIdentity 结构体 { K[]
byte SshKey []byte }
```

图 7：由 Go 载体类型经 PROTOBUF 编译器生成的扁平化 T_{GoProto}）

```
func ToProtoRSAIdentity(input *RSAIdentity) (ProtoRSAIdentity , error) {
    如果输入 == nil {
        返回 ProtoRSAIdentity{}, errors.New("输入␣RSAIdentity␣为空")
    }

    proto := ProtoRSAIdentity{}

    // 转换 RSA 私钥 转换为 PKCS#1 DER 格式
    如果 input.k != nil {
        proto.K = x509.MarshalPKCS1PrivateKey(input.k)
    }

    // 转换 SSH 公钥 转换为线路格式
    如果 input.sshKey != nil {
        proto.SshKey =input.sshKey.Marshal()
    }

    return proto , nil
}
```

图 8：从原始 Go 类型到扁平化 Go 承载类型

直接被调用函数的签名——在翻译提示中附带一小组相关的 Rust API 建议。当翻译给定的 Go 项时，如果该项调用了包 `pkg` 中的 Go API `pkg.f` ，我们会查询与映射目标 crate 相关联的预计算知识库$\mathcal{K}_{\texttt{crate}}$ associated with the mapped target crate `crate`. 查询 Query`pkg.f`会结合 `pkg.f` 的文档和类型签名，并检索排名最高的 API 候选项（在我们的实现中为三个）。每个建议的 API 都会连同其可启用调用的导入路径、类型签名和文档一起注入提示中（图 5）。

3.3 跨语言验证不透明库类型（挑战 2）

为实现端到端验证，我们采用两阶段流水线。首先，通过合成适配器，使值能够跨越 Go–Rust 边界，从而在源语言和目标语言之间建立跨语言互操作性。其次，一旦能够进行跨语言执行，我们就通过差分测试验证每个 Go 函数与其 Rust 翻译之间的输入输出等价性。

通过基于模式的承载表示实现跨语言互操作性。 我们的核心思路是通过一个以单一 Protobuf 模式 T_{Proto} [23]定义的、共享且与语言无关的 中间数据格式（IDF），搭建 Go 与 Rust 之间的桥梁。我们使用大语言模型根据源 Go 类型定义合成此模式：例如，给定`RSAIdentity`（图 2），模型会生成图 6 所示的 IDF 模式。

```
use russh::keys::PublicKey; use rsa::pkcs1::解码 RSA 私钥; fn from_proto_rsa_identity(proto: &ProtoRSAIdentity) ->
Result <RSAIdentity > { let k = proto.k.as_ref() .ok_or_else(| | anyhow::anyhow!("ProtoRSAIdentity 中缺少 RSA 私钥"))?;
let k = RsaPrivateKey::from_pkcs1_der(k) .context(" 无法从 PKCS#1 DER 解析 RSA 私钥 ")?; let ssh_key =
PublicKey::from_bytes(&proto.ssh_key) .context("无法解析 SSH 公钥")?; Ok(RSAIdentity { k, ssh_key , }) }
```

图 9：从原始 Rust 类型到扁平化的 Rust 载体类型

```
导入 "crypto/rsa"

类型 RSAKeyPair 结构体 {
    // 用户定义的包装字段
    identity RSAIdentity
    // 库定义的类型字段
    pubKey *rsa.PublicKey
}
```

图 10：复合 Go 类型RSAKeyPair

```
use rsa::RsaPublicKey; pub struct RsaKeyPair {
pub(crate) identity: RSAIdentity, pub(crate)
pub_key: RsaPublicKey , }
```

图 11：已翻译的Rust 类型 RsaKeyPair

根据 T_{Proto}，Protobuf 编译器会生成特定于语言的载体类型 $T_{\text{Proto}}^{\text{Go}}$ 和 $T_{\text{Proto}}^{\text{Rust}}$，以及将这些载体映射到共享线路格式或从共享线路格式映射回来的（反）序列化代码。具体而言，$T_{\text{Proto}}^{\text{Go}}$ 是一个 Go 结构体类型（例如，图 7 中的 ProtoRSAIdentity），而 $T_{\text{Proto}}^{\text{Rust}}$ 是根据同一模式生成的对应 Rust 类型。这些载体类型在构造上是同构的，并提供了一种共同表示，可据此构建跨语言适配器。

这将跨语言互操作性简化为围绕共享载体表示合成两对转换。在源端，我们需要适配器来映射原程序使用的 Go 类型与 Go 承载类型 $T_{\text{Proto}}^{\text{Go}}$；在目标端，我们同样需要适配器来映射 Rust 承载类型 $T_{\text{Proto}}^{\text{Rust}}$ 与翻译所使用的 Rust 类型。在这两种情况下，载体都充当共同的“交汇点”，值可通过它经由 Protobuf 序列化跨越 Go–Rust 边界。

至关重要的是，这些转换无法通过字段级访问来实现。实际代码经常会封装库定义的对象，而这些对象的内部表示是不透明的，或受库强制实施的不变量约束。例如，`RSAIdentity`（图 2）及其翻译版本 `RsaIdentity` 封装了诸如 `ssh.PublicKey`和 `russh::keys::PublicKey`之类的密码学密钥类型；这些类型只能通过受限的公共 API 构造和查看，而不能直接检查或填充其字段。

使用公共 *API* 合成适配器。 根据从T_{Proto}生成的载体类型，我们通过查询先前

```
1  fn from_proto_rsa_key_pair(proto: &ProtoRSAKeyPair) -> Result <RSAKeyPair > {
2      let identity  =proto.identity.as_ref()
3          .ok_or_else(||      anyhow::anyhow!("ProtoRSAKeyPair 中缺少 RSA 身份"))?;
4      let identity  =  from_proto_rsa_identity(identity)
5          .context("无法解析␣RSA␣身份")?;
6
7      let pub_key_der  =proto.pub_key.as_ref()
8          .ok_or_else(||  anyhow::anyhow!("ProtoRSAKeyPair␣中缺少␣RSA␣公钥"))?;
9      let pub_key  =  RsaPublicKey::from_pkcs1_der(pub_key_der)
10         .context("无法从␣PKCS#1␣DER␣解析␣RSA␣公钥")?;
11
12     Ok(RSAKeyPair {
13         身份,
14         公钥,
15     })
16 }
```

图 12：从扁平化的 PROTOBUF 类型转换为 RUST 类型 RSAKeyPair

构建的库知识库以查找相关的 公共 API，并约束大语言模型仅使用这些 API 组装转换。这对于不透明库类型至关重要，因为无法进行直接的结构转换，正确的构造必须通过库提供的例程完成。例如，Go 到载体适配器（图 8）使用 `x509.MarshalPKCS1PrivateKey`以及接口方法 `ssh.PublicKey.Marshal`。在 Rust 端，载体到 Rust 适配器（图 9）通过 crate 专用解码例程（例如 PKCS#1 解码和公钥解析）重建密钥。

处理复杂类型。在实践中，跨边界传递的值可以具有任意结构，组合多个组件——包括嵌套的用户定义封装类型和不透明的库定义对象。例如，`RSAKeyPair`及其 Rust 翻译版本`RsaKeyPair` （图 10 和图 11）同时包含封装层字段和库类型。为适应这类情况，我们以模块化方式生成载体模式和适配器，并复用针对各个组件预先验证的转换。例如，在图 12（第 4 行）中，`RsaKeyPair`的载体到 Rust 适配器复用了图 9 中针对 `RsaIdentity` 的载体层转换。

模式与载体验证。 由于载体模式（图 6）和适配器（图 8 和图 9）均由大语言模型合成，我们会在将其用于输入/输出等价性测试之前，通过往返检查对其进行验证。具体而言，在 Go 端，我们要求正向/反向适配器 $f_{\text{Go}\to\text{Proto}}$ 和 $f_{\text{Proto}\to\text{Go}}$ （用于原始 Go 类型与 T_{Proto} 之间的转换）在代表性值 v （例如，从单元测试执行中获得的值）上满足 $f_{\text{Proto}\to\text{Go}}(f_{\text{Go}\to\text{Proto}}(v)) = v$ ，以此证明载体保留了足够的信息，能够重建原始 Go 值。Go $\leftrightarrow$ Proto 往返转换成功后，我们通过将单向适配器组合为源类型与目标类型之间的端到端转换，验证完整的跨语言互操作性： $f_{\text{Rust}\to\text{Go}}(f_{\text{Go}\to\text{Rust}}(v)) = v$。通过这些检查是跨越语言边界传输输入和输出的必要前提；满足此前提后，我们才能有意义地测试端到端输入/输出等价性。

如果往返检查失败，我们会在保持翻译不变的情况下重新生成适配器。如果重试次数超过预先设定的上限，我们会重新生成翻译并再次验证。

差分输入输出验证。 一旦确立了函数输入和输出类型的类型互操作性，我们便复用经过验证的适配器，在相同的单元测试输入上（通过载体表示进行映射）执行 Go 源代码和 Rust 翻译，并比较它们可观察到的输出。即使翻译后的代码操作的是库定义的类型，而这些类型原本无法跨越语言边界表示，这也使我们能够在代码仓库级别进行差分测试。

如果输入输出验证失败，我们会冻结翻译后类型、函数签名和适配器，并且只重新生成函数体——最多不超过固定的预算——直到验证成功或预算耗尽。

4 面向 crate 的知识库构建

我们的检索增强生成流水线依赖面向 *crate* 的 检索。给定一个目标 Rust crate，我们会构建一个可查询的知识库，收录其可公开使用的 API 及其有效导入路径，从而在翻译期间实现精准的 API 检索。本节将介绍该知识库的构建过程。

4.1 为 crate 建立 API 索引

对于预先计算出的库映射中的每个目标**crate**，我们都会构建一个特定于 crate 的知识库 $\mathcal{K}_{\text{crate}}$，以支持从源自 Go 文档和签名的自然语言查询中检索相关的 Rust API。

形式化地说，对于每个 crate，我们首先构建一个清单，将每个 API项映射到其文档以及一组可用于调用的导入路径：

$$ApiIndex : \text{API} \to \text{Doc} \times \mathcal{P}(\text{ImportPath}),$$

随后，我们通过可查询的知识库 $\mathcal{K}$ 公开此清单，该知识库可将文本查询映射为按排名排列的 API 条目列表[4]：$\mathcal{K} : \text{String} \to \mathcal{S}(ApiIndex)$。

***Rust*库文档的结构化表示。**我们的$ApiIndex$ 构建算法以 Rust crate 文档的轻量级结构化表示为基础，受到 *rustdoc-json* 格式的启发，并在图 13 中予以形式化。该表示旨在以统一的方式呈现枚举 crate 公共 API 接口所需的信息，并推导出可用于引用每个 API项的有效导入路径集合。

如图 13 所示，一个 Rust crate 由单个根模块 m 组成。每个模块存储：(i) 模块级文档（文档）；(ii) 条目列表 i；以及 (iii) 形式为 `pub use` p 的再导出声明列表。再导出对于 $ApiIndex$ 至关重要，因为它们会在额外的公共路径下公开条目，而这些

[4] 这里 $\mathcal{S}(\cdot)$表示对清单条目的全序排名。

在将自然语言文档中的名称映射到具体 API引用时必须加以考虑。

每个项 i 要么是子模块、类型 t，要么是特征 $\mathbb{T}$，且每个项都带有自己的文档。特征（$\mathbb{T}$）定义一组方法签名 f，而类型（t）则与一组 `impl` 块（ImplBlock）相关联，这些块提供方法，包括特征实现。在图 13 中，我们将 ImplBlock 表示为一个特征实现 `impl` $\mathbb{T}$ `for` t 及其方法集 f；尽管此定义经过简化，我们的实现实际上能够处理所有 Rust `impl` 形式。最后，路径 p 编码了形如$m(::n)^* :: i$的完全限定名称，这些名称是 $ApiIndex$ 使用的规范导入路径。

模块 $m := \text{文档} \times \overline{i} \times \overline{\texttt{pub use } p}$
Item $i := t \mid \mathbb{T} \mid f$
Type $t := \text{文档} \times \overline{\text{ImplBlock}}$
特征 $\mathbb{T} := \text{文档} \times \overline{f}$

ImplBlock $:= \text{为 } \mathbb{T} \text{ 实现 } t \times \overline{f}$
函数 $f := \text{文档}$
Path $p := m(::m)^* :: i$

图 1 3：Rust 库文档的结构表示。

Rust ApiIndex 构建。 我们利用 Rust 库文档的结构化表示，为 Rust crate 构建 $ApiIndex$。为此，我们系统枚举其可公开使用的 API 接口，以及 支持调用的 导入路径。这需要考虑深层模块层级、`pub use`再导出（包括来自传递依赖的再导出），以及特征作用域内的方法可调用性。因此，我们的算法遍历公共模块树，利用依赖项的预计算清单解析内部再导出与外部重导出，并提取类型、trait 和方法。对于特征方法，我们最初记录占位特征引用，随后在最终修订阶段将其改写为具体导入路径。我们表明，所有生成的路径在 Rust 可见性规则下均有效。完整细节（包括算法与证明）见附录 A。

Go ApiIndex 构建。 值得注意的是，在合成Go 与 Rust 之间的胶水代码时（见第 5.1 节），我们还需要 Go库的知识库。构建$ApiIndex$ 供 Go 使用十分直接，因其类型和包系统简单——引用 API 只需库名和 API 名称（例如，`ssh.PublicKey` 见图 2）。这可通过网页抓取或类似方法实现。在本工作中，我们解析 `go doc` 命令的输出，该命令提供全面的库文档。

4.2 构建知识库

基于每个 crate 的清单 $ApiIndex$，我们构建知识库 $\mathcal{K}$，用于从自然语言查询中检索相关API 候选项。具体而言， $\mathcal{K}$ 会排序

条目按与查询 Q 的相关性（例如，源自 Go 文档和签名），并返回最匹配的结果及其可用于调用的导入路径。

两阶段检索与重排序。 对于真实场景中的 crate，直接使用大语言模型对完整的 $ApiIndex$ 进行重排序并不可行，因为这类 crate 可能会公开数千个 API。因此，我们采用标准的两阶段信息检索流水线：先由高效的第一阶段检索器进行检索，再由准确性更高的重排序器处理。

给定查询 Q，我们首先采用稠密双编码器检索器，按照稠密段落检索（DPR）[31]的方式，根据 API 条目与其文档之间的语义相似度进行评分，并选取一小组候选项（前 k 个；在我们的实现中，$k = 35$ 之后的召回率提升已趋于平缓）。然后，我们使用更强大的交叉编码器模型对这些候选项重新排序，方式类似于基于 BERT 的重排序 [39]。最后，我们使用基于大语言模型的列表式重排序器，得到最终排序 [46]。

5 验证翻译

本节将构建一个验证框架，用于检查存在库依赖时函数翻译的等价性。如第 2 节所述，即使生成的代码能够编译，验证 Go 到 Rust 的翻译仍然具有挑战性，因为这两种语言的值表示互不兼容，且广泛使用库定义类型。为了支持端到端验证，我们采用两阶段流水线：首先，通过合成适配器，在 Go–Rust 边界安全地序列化值，从而建立跨语言互操作性；然后，利用这些经过验证的适配器，通过对单元测试派生的输入进行差分测试，检查每个 Go 函数与其 Rust 翻译之间的输入输出等价性。

5.1 跨语言互操作性

我们的目标是实现用于验证的跨语言执行：给定一个类型为 T_{Go} 的 Go 值，我们必须构造对应的、类型为翻译后类型T_{Rust} 的 Rust 值，反之亦然，即能够还原出 Go 值。我们将其建模为合成一对 跨语言适配器

$$\mathcal{A}_T : T_{\text{Go}} \to T_{\text{Rust}} \qquad \mathcal{A}_T^{-} : T_{\text{Rust}} \to T_{\text{Go}},$$

其中， $\mathcal{A}_T$ 将 Go 单元测试输入/输出提升为 Rust 中的值，以便进行后续差分测试； $\mathcal{A}_T^{-}$ 支持往返验证。特别是，组合$\mathcal{A}_T^{-} \circ \mathcal{A}_T$ 应保留可观察到的 Go 值，从而证明 (i) 翻译后的 Rust 类型不会丢弃语义相关的信息，以及 (ii) 这些适配器在真实输入上能够保持行为不变。

直接生成 $\mathcal{A}_T$ 和 $\mathcal{A}_T^{-}$ 很困难，因为 Go 和 Rust 的值具有不兼容的表示形式，而库定义类型通常是不透明的（具有私有字段），只能通过公共 API 操作。为了解决这个问题，我们引入了一种基于 Protobuf 的共享中间表示 T_{Proto}，用作跨越 Go–Rust 边界的语言无关载体。

Protobuf 模式生成。 接下来，我们将介绍如何从复合源 Go 类型 T_{Go}合成载体模式 T_{Proto}

一个关键区别在于语言或库定义的 复合类型与用户定义的 复合类型。对于前者，给定复合类型 T_{Go}，大语言模型会为其每个字段选择合适的 Protobuf 编码。Go 原始类型（例如，`int32`、`string`、`bool`）会直接映射到相应的 Protobuf 标量类型。相比之下，复杂或不透明的字段——尤其是具有私有内部表示的库定义类型——会保守地表示为字节数组。采用这种字节数组回退方案，可确保即使无法访问值的内部结构，或其内部结构没有自然对应的 Protobuf 类型，这些值仍可跨越语言边界传输。

对于包含嵌套用户定义依赖的用户定义复合类型，我们会按照依赖顺序以组合方式生成模式。大语言模型会逐步为引用了先前已处理类型的类型构建模式。在为引用另一用户定义类型 T' 的类型 T 生成模式时，我们会指示大语言模型复用现有的 T' Protobuf 定义，以确保各模式间的一致性并避免重复声明。

这种对用户定义复合类型结构的公开，对于后续阶段至关重要：它使我们能够合成胶水代码，将字段级转换组合起来，并在 Go、Rust 与载体表示之间进行映射时，系统地重用现有库 API。相比之下，库定义类型的内部结构通常对客户端不透明，因此会采取保守处理，通过基于 API 的适配器而非字段级访问来处理。

高层胶水代码生成。根据Protobuf 模式，Proto-buf 编译器会生成特定语言的载体类型 $T^{\text{Go}}_{\text{Proto}}$ 以及 $T^{\text{Rust}}_{\text{Proto}}$底层序列化和反序列化例程，用于将值编码为共享线路格式（并从中解码值）。这些由编译器生成的类型和例程构成了轻量级互操作层，无需显式操作外部函数接口。从概念上说，该模式 T_{Proto} 定义了一种语言无关载体，并在Go 和Rust中分别具体实现。因此，我们将 T_{Proto}视为一种同时存在于两种语言中的类型，并将互操作问题简化为合成高层胶水代码，以连接 T_{Proto} 与源类型 T_{Go}及翻译后类型 T_{Rust}。

固定 T_{Proto} 后，跨语言互操作性归结为生成语言本地胶水代码，在每个 T_{Lang} 和 T_{Proto}之间进行转换：

$$\mathcal{A}_{T_{\text{Lang}}} : T_{\text{Lang}} \to T_{\text{Proto}} \qquad \mathcal{A}^{-}_{T_{\text{Lang}}} : T_{\text{Proto}} \to T_{\text{Lang}},$$

其中 Lang $\in$ {Go，Rust}。这些适配器在丰富的语言特定数据结构与共享的 Protobuf 编码之间架起桥梁。

从设计上看， T_{Proto} 很轻量，仅包含 Protobuf 原生基元，例如标量、枚举、字节数组和浅层聚合类型。相比之下， T_{Lang} 类型可能深度嵌套、包含用户定义的结构，并依赖只能通过公共 API 访问的库类型。

我们使用大语言模型合成胶水函数，并向其提供 T_{Lang} 和 T_{Proto}的定义。对于复合类型，我们会指示模型生成胶水

代码时采用组合式方法，尽可能复用现有转换，并在必要时参考检索到的库指南：

1. **复用现有适配器**。如果 T_{Proto} （或 T_{Lang}）依赖嵌套类型 U，我们会复用先前为 U 生成的适配器，以避免重复转换逻辑，并确保整个代码仓库的一致性。2. **知识库引导的库转换**。如果 T_{Lang} 包含库lib中的类型，我们会将从相应知识库 $\mathcal{K}_{lib}$（查询$T_{\text{Lang}} \leftrightarrow T_{\text{Proto}}$）检索到的 API 补充到提示中。查询$T_{\text{Lang}} \leftrightarrow T_{\text{Proto}}$包含 T_{Lang} 和 T_{Proto} 的定义，以引导模型使用可调用的构造函数、访问器和（反）序列化例程。

尽管有这些指导，大语言模型生成的胶水代码仍容易出错：模型可能会选择不合适的 API、引入类型不匹配，或生成虽能编译、却在运行时违背语义的代码。因此，在进行下游等价性测试之前，我们会先通过往返检查验证适配器，具体如下所述。

往返验证。由于载体模式 T_{Proto}、翻译后的 Rust 类型 T_{Rust}以及胶水函数 $\mathcal{A}_{T_{\text{Lang}}}$ 和 $\mathcal{A}^{-}_{T_{\text{Lang}}}$ 均由大语言模型合成，因此在将其用于下游等价性检查之前，我们会明确验证它们是否保留了信息。我们以源类型 T_{Go} 为基准，分两个阶段进行验证：首先验证 Go $\leftrightarrow$ Proto 转换，然后才验证完整的 Go $\leftrightarrow$ Rust 互操作性。

*Go*端往返转换。 在第一阶段，我们验证 Go 端胶水代码能否通过载体表示往返转换值而不造成损失：

定义 1（Go端往返性质）。 给定 *Go* 类型 T_{Go} 以及变换 $\mathcal{A}_{T_{Go}}$：$T_{Go} \to T_{\text{Proto}}$ 和 $\mathcal{A}^{-}_{T_{Go}}$： $T_{\text{Proto}} \to T_{Go}$，以下性质成立：

$$\forall v. \quad \mathcal{A}^{-}_{T_{Go}}\big(\mathcal{A}_{T_{Go}}(v)\big) = v.$$

满足定义 1意味着前向嵌入 $\mathcal{A}_{T_{\text{Go}}}$ 在经过测试的值上是单射，这表明生成的模式 T_{Proto} 涵盖了 T_{Go} 中所有语义相关的信息（即载体没有引入信息损失或意外规范化）。

通常无法确立这一全称量化性质，因此我们使用从现有单元测试中提取的具体值 v 对其进行近似验证。失败表明载体模式不足，或 Go 端胶水代码有误，并会触发重新生成两者 T_{Proto} 以及 Go $\leftrightarrow$ Proto 适配器。

完整往返转换。 Go $\leftrightarrow$ Proto 往返转换成功后，我们通过组合单向适配器，在源类型和目标类型之间构建端到端变换，以验证完整跨语言互操作性：

$$\mathcal{A}_T = \mathcal{A}^{-}_{T_{\text{Rust}}} \circ \mathcal{A}_{T_{\text{Go}}} \qquad \text{and} \qquad \mathcal{A}^{-}_T = \mathcal{A}^{-}_{T_{\text{Go}}} \circ \mathcal{A}_{T_{\text{Rust}}}.$$

然后，我们验证相应的往返性质：

定义 2（完整往返属性）。给定一个Go 类型 T_{Go} 以及变换 $\mathcal{A}_T$：$T_{Go} \to T_{Rust}$ 和 $\mathcal{A}_T^-$： $T_{Rust} \to T_{Go}$，以下性质成立：

$$\forall v. \quad \mathcal{A}_T^-(\mathcal{A}_T(v)) = v.$$

与之前一样，我们通过在有限的值集上进行测试来近似定义 2。由于第一阶段已确立 Go $\leftrightarrow$ Proto 转换的正确性，因此本阶段出现的故障可归因于 Rust 端产物（翻译后类型 T_{Rust} 或 Rust $\leftrightarrow$ Proto 胶水代码）。因此，当完整的往返检查失败时，我们只重新生成 Rust 端适配器，避免不必要地重新计算 T_{Proto} 以及已经验证的 Go 端胶水代码。

5.2 输入输出等价性的验证

在确立互操作性（第 5.1 节）后，我们通过差分测试验证可观察行为。给定一个 Go 函数 f 及其由大语言模型生成的 Rust 翻译版本 g:

$$f : I_{\text{Go}} \to O_{\text{Go}} \qquad g : I_{\text{Rust}} \to O_{\text{Rust}}.$$

为简洁起见，本节中我们假设输入和输出均为单一类型。由于输入和输出可能涉及不透明的库定义类型，我们通过比较借助第 5.1 节中的适配器执行的结果来验证输入输出等价性；这些适配器负责在 Go–Rust 边界传输值:

定义 3（输入输出等价性）。给定一个 *Go* 函数 f 和 *Rust* 翻译版本 g，以及上一步生成的转换，

$$\mathcal{A}_I : I_{Go} \to I_{Rust} \qquad \mathcal{A}_O : O_{Go} \to O_{Rust}$$

则当 $\forall v. \mathcal{A}_O(f(v)) = g(\mathcal{A}_I(v))$时，$f$与 g具有输入输出等价性。

观测输入上的输入输出等价性。 我们假设源代码仓库包含单元测试，其执行会为 f产生具体的输入输出示例（i，o）。我们将这些示例视为行为规范，并通过检查 g 在观测输入上与 f 相匹配来验证输入输出等价性。具体而言，令 i 表示从单元测试运行中提取的Go输入集合；当定义3对每个 $v \in i$ 均成立时（对于 i，（i，o）中的输入），我们报告（f，g）的输入输出等价性。

对保证的解读。 我们的检查并不能确立 f 与 g之间的完全语义等价性；相反，它们提供证据表明，翻译在观测输入上保持了行为。为了在Go层面解读这一点，我们将逆适配器 $\mathcal{A}_O^-$ 应用于Rust输出，从而得到：

引理 1（在观测输入上的 Go 级行为一致性）。假设输出类型 O 对 $\{f(v) \mid v \in i\}$ 中的所有值均满足完整往返属性（定义 *2*），并假设所有 $v \in i$ 均满足输入输出等价性。那么，对于每个$v \in i$,

$$f(v) \;=\; \mathcal{A}_O^-(g(\mathcal{A}_I(v))),$$

也就是说，对于观测输入，f 产生的输出与函数 $\mathcal{A}_O^- \circ g \circ \mathcal{A}_I$相同。

表1：基准测试说明。代码行数不包括注释和空行（gocloc [25] 代码计数）。倒数第二列报告项目调用的外部函数的代码行数总计，每个函数只计一次。最后一列显示项目中使用的不同库 API 的数量。

基准测试依赖	领域	星标/分叉数Go	代码行数（无库）	Go 代码行数（+ 库）	外部库条目
pbkdf2 [58]	密钥派生	11/5	270	2996	29
easy-scrypt [5]	Crypto	25/3	196	5484	29
simple-scrypt [16]	Crypto	201/27	322	10517	10
argon2-hashing [7]	密码	25/4	498	6485	9
持续时间 [43]	时间解析	32/2	166	12041	9
age [20]	Crypto	21.1k/609	51	4103	38

证明。 任取 $v \in i$。根据输入输出等价性（定义 3），我们有 $\mathcal{A}_O(f(v)) = g(\mathcal{A}_I(v))$。根据输出的完整往返属性（定义 2）， $\mathcal{A}_O$ 和$\mathcal{A}_O^-$ 在 $f(v)$ 上有定义，并满足 $\mathcal{A}_O^-(\mathcal{A}_O(f(v))) = f(v)$。对上述等式两边应用 $\mathcal{A}_O^-$，得到 $f(v) = \mathcal{A}_O^-(g(\mathcal{A}_I(v)))$。由于 v 是任意的，该结论对所有 $v \in i$ 均成立。

命题1表明，即使涉及不透明的库类型，Rust 实现也可以在观测输入上替代 Go 函数。

6 实验评估

在本节中，我们针对以下研究问题给出结果。

– **RQ1：（总体有效性）**我们的方法能否翻译大量使用外部库的完整项目？我们的方法能否验证以不透明的外部库类型作为参数的函数的功能等价性？ – **RQ2：（消融）**我们的检索增强生成与 Rust 导入解析方法对翻译成功的贡献有多大？ – **RQ3：（基线）**我们的方法与**Oxidizer** [62]相比如何?

6.1 实验设置

实现。我们基于工具**Oxidizer** [62], 构建了我们的方法。Oxidizer 是一款最先进的代码转换工具。我们的修改为**Oxidizer**的实现增加了 1580 行 Go、1245 行 Rust 和 2585 行 Python 代码。我们将新实现命名为**CrossCrate**，其使用方式与**Oxidizer**相同：输入一个 Go 项目和若干用于控制重试预算的超参数，输出该项目的 Rust 翻译版本。

表 2：翻译结果。RAG 负载是翻译任务涉及的所有 Rust crate 中可检索 API 条目的总数，计算方式为所构建的每个 crate 知识库大小之和，并近似表示 RAG 候选搜索空间的大小。

		Full		无 RAG		无导入	
基准测试	RAG编译 load	完整/依赖	等价性 完整/依赖	编译。 完整/依赖	等价性 完整/依赖	编译。 完整/依赖	等价性 完整/依赖
pbkdf2 [58]	341	100/100	100/100	0/0	0/0	0/0	0/0
easy-scrypt [5]	1382	100/100	100/100	33/0	33/0	33/0	33/0
simple-scrypt [16]	1388	100/100	100/100	60/0	60/0	60/0	60/0
argon2-hashing [7]	139	100/100	100/100	80/0	80/0	80/0	80/0
持续时间 [43]	1341	100/100	100/100	75/0	75/0	75/0	75/0
age [20]	3543	75/75	80/80	0/0	0/0	0/0	0/0
平均值	1356	95.83/95.83	95.83/95.83	41.33/0	41。33/0	41。33/0	41。33/0

大语言模型与超参数。所有实验均采用与原始**Oxidizer** 工作相同的超参数设置 [62] 。我们在所有实验中都使用 Claude Sonnet 3.5 [8]作为大语言模型。它在编码任务上的表现与其他最先进的大语言模型相近，因此我们认为，我们的方法也适用于其他大语言模型。为支持可复现性，我们的实现还支持重放记录的大语言模型响应，而无需查询真实的大语言模型。

基准测试。我们在一组精选的真实世界Go基准测试上评估我们的方法，这些基准测试取自 GitHub。我们的筛选标准如下：(1) 基准测试涉及非平凡的库 API，使检索增强生成面临庞大的候选搜索空间；(2) 它们涵盖不同的依赖领域，包括密码学密钥派生、内存困难型密码哈希，以及时间和时长解析；(3) 它们得到积极维护，并应用于生产系统。依赖领域的这种多样性使我们能够研究该方法在根本不同的 API 和依赖结构下的表现，范围涵盖对安全至关重要、参数丰富的密码学库，以及具有复杂控制流和类型级约束的领域专用解析库。

具体而言，我们的基准测试套件由以下库组成。表1总结了这些基准测试、其依赖领域以及受欢迎程度（星标数和复刻数）。

– pbkdf2 [58]：一个密码学密钥派生库，实现了 PBKDF2 算法，可通过盐值和迭代次数从密码派生安全密钥。此基准测试涵盖标准密码学 API 和带参数的安全原语。– easy-scrypt [5] 和 simple-scrypt [16]：提供基于 scrypt 的密码哈希功能的内存困难型密钥派生库。这些基准测试着重考察安全关键代码的翻译，其中涉及复杂的参数验证、内存成本配置以及性能敏感型 API。

– argon2-hashing [7]：一个基于 Argon2 算法的密码哈希库，体现了现代内存困难型密码哈希的特点，具有结构化的参数类型和严格的 API 级约束。– duration [43]：一个时间处理库，用于解析和格式化易读的时长字符串，涵盖领域专属的解析逻辑，以及数值、时间和字符串处理 API 之间的交互。– age [20]：一款简单、现代且安全的文件加密工具，采用小型显式密钥并支持后量子技术。我们从该库中提取了具有复杂库依赖的函数和类型，同时剔除了包含 **Oxidizer** 不支持功能的片段（例如涉及文件输入输出的片段）。

6.2 结果

我们在每个基准测试上运行 **CrossCrate**，并在表 2 的 **Full** 列中报告结果。我们报告两个关键指标：**% Comp.** 列中的编译率，以及 **% Equiv.** 列中的输入输出等价率。这两个指标均按可编译或等价的函数所占百分比计算。我们针对所有函数（**full**）以及使用库依赖项的函数子集（**dep**）报告这些指标。

RQ1（总体有效性）。表 2 的结果表明，**CrossCrate**确实能够有效转换使用外部库的项目，并验证使用外部库类型作为参数的函数在功能上等价。**CrossCrate** 成功转换并验证了我们所有基准测试中 100% 的函数。

RQ2（消融实验）。接下来，我们评估知识库和导入解析在我们的检索增强生成方法中的作用。我们以两种模式运行 **CrossCrate**：一种同时禁用知识库查询和导入解析，另一种仅启用知识库查询。结果分别显示在表 2 的 **No RAG** 列和 **No Imports** 列中。结果表明，如果缺少检索增强生成方法的这两个组件，大语言模型就无法为具有库依赖项的函数生成可编译（且等价）的代码，整体编译率和等价率也会显著下降。这些发现表明，导入解析是我们流水线中必不可少的组件，而仅进行 API 映射、却不提供使调用可用的导入项，在实际应用中并不足够。

RQ3（基线）。**Oxidizer** 会为翻译硬编码目标 crate；然而，这对复杂库帮助有限——其表现与**无 RAG** 消融实验完全相同，在使用外部 API 的函数上，编译和等价性成功率均为 0%。

7 相关工作

代码翻译验证。代码翻译验证最初是在编译器领域中研究的 [36, 45]，，直到最近才受到

我们领域内的广泛关注 [42, 48, 49, 53], ，也就是源到源代码转换。早期研究提出了基于规则的翻译系统 [1, 3, 11, 35], ，其中许多专注于 C语言到 Rust 的转换 [17, 21, 26, 29, 34, 50, 63], 。这类系统在理论上通过构造保证正确，但通常会生成不符合惯用写法的翻译结果，或面向特定领域 [6, 30, 47, 54, 61, 65]。近期研究则利用大语言模型生成候选翻译，再结合等价性验证系统。验证分为两个层级：系统级 [33] 或函数级 [4,9,10,18,28,32,51,60,62,66]。系统级验证通过执行文本界面（例如命令行界面）来完成。系统级验证研究假定源程序与翻译结果的文本界面相同，因此没有解决我们所处理的挑战；此外，这些方法无法验证不提供文本界面的库翻译。函数级验证研究通常假定函数输入类型在结构上相似。SACTOR [66] 是唯一不作此假定的研究，但其方法无法处理不透明类型，并且依赖于针对 C语言和 Rust 手工编写的规则。

未经验证的代码翻译。许多代码翻译研究只关注生成最佳候选翻译，却没有提出自动化验证系统 [12, 14, 27, 38, 42, 52, 55–57, 59, 64]。这些研究转而使用手写测试来评估翻译的正确性，与我们的工作正交。**基于大语言模型的外部库代码生成**近期一项研究表明，当源代码大量使用外部库时，大语言模型在代码翻译中的表现会下降 [24]。一项与我们同期开展的研究提出了与我们类似的库映射方法，通过文档查找目标语言中的等效外部库 [32], ，但没有像我们一样解决 Rust 特有的挑战。此外，也有研究在代码翻译之外探讨基于大语言模型的外部库代码生成 [15], ，但这些研究并未尝试构建库映射。

跨语言差分测试与验证。除了代码翻译领域之外，也有大量关于差分测试与验证的研究 [22, 37, 40, 41, 44], ，但这些研究假设两个受测实体的接口完全相同，因此没有解决我们面临的一项核心挑战。

8 结论

我们提出了一种面向库、由检索增强生成指导的 Go 到 Rust 翻译框架。该框架利用精选知识库中的公开且可从 crate 根访问的条目，将外部 Go API 映射到 Rust，随后分两步验证结果：先进行跨语言往返，再进行差分输入输出。在具有外部依赖项的项目中，该方法使依赖外部依赖项的函数也能够完成翻译，从而同时提升了编译成功率和等价性成功率（平均约提升至原来的 2 倍）。

参考文献

1. C 到 Go 转换器，https://github.com/gotranspile/cxgo

2. 彻底消除内存安全漏洞（DARPA），https: // www.darpa.mil/news-events/2024-07-31a 3. Sharpen – 自动化 Java->C# 转换，https://github.com/mono/sharpen 4. Abid, M.S., Pawagi, M., Adhikari, S., Cheng, X., Badr, R., Wahiduzzaman, M., Rathi, V., Qi, R., Li, C., Liu, L., 等：GlueTest：通过语言互操作性测试代码转换。载于：2024年IEEE国际软件维护与演化会议（ICSME）。第 612–617 页。IEEE（2024年）5. agnivade：GitHub - agnivade/easy-scrypt：适用于 Go 的简易 scrypt 绑定。https: //github.com/agnivade/easy-scrypt（2026 22 6 2026 01），访问日期：- -。Ahmad, M.B.S., Ragan-Kelley, J., Cheung, A., Kamil, S.：自动将图像处理库转换为 Halide。载于：ACM Trans. Graph. 第 38 卷。ACM（2019年11月）。https://doi.org/10.1145/3355089.3356549，https://doi.org/10.1145/ 3355089.3356549 7. andskur：GitHub - andskur/argon2-hashing：Go 中的 Argon2 密码哈希。https: // github.com/andskur/argon2 22 8 2026 2026 01 2024-hashing（），访问日期：- -。Anthropic：Claude Code（），https://www.anthropic.com/news/ claude-3-5-sonnet，发布方：网站 9. Bai, Y., Palit, T.：RustAssure：对经大语言模型转译的C到Rust代码进行差分符号测试。arXiv预印本 arXiv:2510.07604（2025年）10. Bhatia, S., Qiu, J., Hasabnis, N., Seshia, S.A., Cheung, A.：借助大语言模型实现经验证的代码转译。载于：神经信息处理系统进展。第 37 卷，第 41394–41424 页（2024年），https://proceedings.neurips.cc/paper_files/paper/ 2024/file/48bb60a0c0aebb4142bf314bd1a5c6a0-Paper-Conference.pdf 11. Buddrus, F., Schödel, J.：Cappuccino——C++ 到 Java 的翻译器。载于：1998年ACM应用计算研讨会论文集。第 660–665 页（1998年）12. Cai, X., Liu, J., Huang, X., Yu, Y., Wu, H., Li, C., Wang, B., Yusuf, I.N.B., Jiang, L.：RustMap：通过程序分析和大语言模型实现项目级C到Rust迁移。arXiv预印本 arXiv:2503.17741（2025年）13. DARPA：TRACTOR：将所有C代码转换为Rust（2025年），https://www.darpa.mil/ research/programs/translating-all-c-to-rust，发布方：DARPA 网站 14. Dehghan, S., Sun, T., Wu, T., Li, Z., Jabbarvand, R.：将大规模C代码库转换为惯用Rust。arXiv预印本 arXiv:2511.20617（2025年）15. Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., Lou, Y.：评估大语言模型的类级代码生成能力。载于：IEEE/ACM第46届国际软件工程会议论文集。第 1–13 页（2024年）16. elithrar：GitHub - elithrar/simple-scrypt：适用于 Go 的简单 scrypt 封装器。https: // github.com/elithrar/simple-scrypt（2026 22 17 2026 01），访问日期：- -。Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.：将 C 转换为更安全的 Rust。《ACM程序设计语言会刊》5（OOPSLA）（2021年10月）。https://doi.org/10.1145/ 3485498，3485498 18 10 1145 https://doi.org/ . / . Eniser, H.F., Wüstholz, V., Christakis, M.：自动测试代码转换模型的功能属性。载于：AAAI人工智能会议论文集。第 38 卷，第 21055–21062 页（2024年）19. Eniser, H.F., Zhang, H., David, C., Wang, M., Paulsen, B., Dodds, J., Kroening, D.：迈向大语言模型驱动的真实世界代码转换：一项转换为 Rust 的研究。arXiv预印本 arXiv:2405.11514（2024年）20. Filippo Valsorda：GitHub - filosottile/age：一种简单、现代且安全的加密工具（及 Go 库），使用小型显式密钥，无配置选项，并支持 UNIX 风格的组合。https://github.com/FiloSottile/age（2026年），访问日期：2026-01-22

21. Fromherz, A., Protzenko, J.：《将 C 编译为安全 Rust：形式化》（2024），https: //arxiv.org/abs/ 2412 15042 _ 2412 15042 . , eprint: . . Garzella, J.J., Baranowski, M., He, S., Rakamarić, Z.：《利用编译器中间表示进行多语言与跨语言验证》。载于：验证、模型检测与抽象解释：第 21 届国际会议 VM- CAI 2020。第 90–111 页。Springer，柏林，海德堡（2020）。https://doi.org/10. 1007 978 3 030 39322 9_5 10 1007 978 3 030 39322 9_5 / - - - - , https://doi.org/ . / - - - - . 谷歌：Protocol Buffers。https://protobuf.dev/ ( )，访问日期：- -26 24. Gu, X., Chen, M., Lin, Y., Hu, Y., Zhang, H., Wan, C., Wei, Z., Xu, Y., Wang, J.：《大语言模型在领域特定代码生成中的有效性》。ACM 软件工程与方法论汇刊 **34**(3), 1–22 (2025) 25. Hattori, H.：gocloc：一个快速的 Go 代码行计数工具（2015），https://github. com/hhatto/gocloc 26. Hong, J., Ryu, S.：《不写入，而是返回：在 C 到 Rust 翻译中用代数数据类型替换输出参数》。ACM 编程语言论文集 **8**(PLDI) （ 2024 年 6 月 3656406 27 10 1145 3656406, 10 1145）。https://doi.org/ . / https://doi.org/ . / . Hong, J., Ryu, S.：《使用大语言模型进行类型迁移式 C 到 Rust 翻译》。实证软件工程 **30**(1), 3 (2025) 28. Ibrahimzada, A.R., Ke, K., Pawagi, M., Abid, M.S., Pan, R., Sinha, S., Jabbarvand, R.：AlphaTrans：一种用于代码仓库级代码翻译与验证的神经符号组合方法。载于：ACM 软件工程论文集。第 2 卷，第 2454–2476 页。ACM (2025) 29. Immunant：c2rust：将 C 代码迁移至 Rust，https://github.com/immunant/c2rust 30. Kamil, S., Cheung, A., Itzhaky, S., Solar-Lezama, A.：《模板计算的经验证提升》。载于：第 37 届 ACM SIGPLAN 编程语言设计与实现会议论文集。第 711–726 页。PLDI ’16，ACM (2016)。https://doi.org/10.1145/2908080.2908117, https://doi.org/10.1145/ 2908080.2908117 31. Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., Yih, W.：《面向开放域问答的稠密段落检索》。载于：Webber, B., Cohn, T., He, Y., Liu, Y.（编）《2020 年自然语言处理实证方法会议论文集》，EMNLP 2020，20，2020 年 11 月 16 日 6769 6781 2020Online。第 – 页。计算语言学协会（）。https: //doi.org/10.18653/V1/2020.EMNLP-MAIN.550, https://doi.org/10.18653/v1/ 2020 550 32 .emnlp-main. . Ke, K., Ibrahimzada, A.R., Pan, R., Sinha, S., Jabbarvand, R.：《推进代码仓库级代码翻译中的自动化隔离验证》。arXiv 预印本 arXiv:2511.21878 (2025) 33. Li, T., Li, R., Wang, B., Paulsen, B., Mathur, U., Saxena, P.：《用于 C 到 Rust 翻译的对抗式智能体协作》。arXiv 预印本 arXiv:2510.03879 (2025) 34. Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J.R., Hassan, A.E.：《我们信赖 Rust：一种将不安全 C 转译为更安全 Rust 的转译器》。载于：ACM/IEEE 第 44 届国际软件工程大会配套论文集。第 354–355 页 (2022) 35. Malabarba, S., Devanbu, P., Stearns, A.：MoHCA-Java：支持将 C++ 转换为 Java 的工具。载于：第 21 届国际软件工程大会论文集。第 650–653 页（1999）36. Necula, G.C.：《面向优化编译器的翻译验证》。载于：ACM SIGPLAN 2000 编程语言设计与实现会议论文集。第 83–94 页。PLDI ’00，ACM (2000)。https://doi.org/10.1145/ 349299.349314, https://doi.org/10.1145/349299.349314

37. Nilizadeh, S., Noller, Y., Păsăreanu, C.S.：DifFuzz：用于侧信道分析的差分模糊测试。载于：第 41 届国际软件工程大会论文集。第 176–187 页。ICSE '19，IEEE 出版社（2019）。https://doi.org/10. 1109 00034 38 2019 00034, 10 1109 2019 2 /ICSE. . https://doi.org/ . /ICSE. . . Nitin, V., Krishna, R., Valle, L.L.d., Ray, B.：C SaferRust：运用神经符号技术将 C 项目转换为更安全的 Rust。arXiv 预印本 arXiv:2501.14257 (2025 14257 39 2501)，https://arxiv.org/abs/ . . Nogueira, R., Cho, K.：基于 BERT 的段落重排序。CoRR **abs/1901.04085** (2019 04085 40 1901 04085, 1901)，http://arxiv.org/abs/ . arXiv: . . Noller, Y., Păsăreanu, C.S., Böhme, M., Sun, Y., Nguyen, H.L., Grunske, L.：HyDiff：混合差分软件分析。载于：ACM/IEEE 第 42 届国际软件工程大会论文集。第 1273–1285 页。ICSE '20，ACM (2020)。https://doi.org/10.1145/3377811.3380363, https://doi.org/10. 1145/3377811.3380363 41. Palikareva, H., Kuchta, T., Cadar, C.：《疑云：检测软件版本间差异的测试》。载于：第 38 届国际软件工程大会论文集。第 1181–1192 页。ICSE '16，ACM (2016)。https://doi.org/ 10 2884845 42 1145 2884781 2884845, 10 1145 2884781 . / . https://doi.org/ . / . . Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R.：《迷失在翻译中：研究大语言模型在代码翻译过程中引入的缺陷》。载于：IEEE/ACM 第 46 届国际软件工程会议论文集。ICSE '24，ACM (2024)。https://doi.org/10.1145/3597503.3639226, https://doi.org/10. 1145/3597503.3639226 43. peterhellberg：GitHub - peterhellberg/duration：Go 中的时长解析与格式化。https://github.com/peterhellberg/duration (2026 22 44 2026 01)，访问日期：- -。Petsios, T., Tang, A., Stolfo, S., Keromytis, A.D., Jana, S.：NEZHA：高效的领域无关差分测试。载于：2017 年 IEEE 安全与隐私研讨会（SP）。第 615 27 45 632 2017 10 1109 2017 – ( )。https://doi.org/ . /SP. . . Pnueli, A., Siegel, M., Singerman, E.：《翻译验证》。载于：系统构造与分析工具和算法。LNCS，第 1384 卷，第 151–166 页。Springer (1998)。https://doi.org/10.1007/BFB0054170, https://doi.org/ 10.1007/BFb0054170 46. Pradeep, R., Sharifymoghaddam, S., Lin, J.：RankZephyr：高效、稳健的零样本列表式重排序易如反掌！CoRR**abs/2312.02724** (2023)。https://doi.org/10.48550/ARXIV.2312.02724, https://doi.org/10.48550/arXiv.2312. 02724, 02724 47 2312arXiv: . . Qiu, J., Cai, C., Bhatia, S., Hasabnis, N., Seshia, S.A., Cheung, A.：Tenspiler：一种用于张量运算的基于验证提升的编译器。载于：Aldrich, J., Salvaneschi, G.（编）《第 38 届欧洲面向对象编程大会》（ECOOP 2024）。莱布尼茨国际信息学论文集（LIPIcs），第 313 卷，第 32:1–32:28 页。施洛斯·达格斯图尔—莱布尼茨信息学中心（2024）。https://doi.org/10. 4230/LIPIcs.ECOOP.2024.32, https://drops.dagstuhl.de/entities/document/ 10.4230/LIPIcs.ECOOP.2024.32 48. Roziere, B., Lachaux, M.A., Chanussot, L., Lample, G.：《编程语言的无监督翻译》。载于：第 34 届神经信息处理系统国际会议论文集。NIPS '20，Curran Associates Inc.，美国纽约州雷德胡克（2020）49. Roziere, B., Zhang, J.M., Charton, F., Harman, M., Synnaeve, G., Lample, G.：《利用自动化单元测试进行无监督代码翻译》（2022），https: //arxiv.org/abs/2110.06773, _eprint: 2110.06773

50. Sharp, J.：Corrode：从 C语言 到 Rust 的自动语义保持翻译，https://github.com/jameysharp/corrode 51. Shetty, M.、Jain, N.、Godbole, A.、Seshia, S.A.、Sen, K.：Syzygy：利用大语言模型和动态分析进行双重代码测试的 C 到（安全）Rust 翻译（2024），https：//arxiv.org/abs/ 2412 14234 _ 2412 14234 。，电子印本：。。Shiraishi, M.、Cao, Y.、Shinagawa, T.：SmartC Rust：利用大语言模型进行迭代式、反馈驱动的 C 到 Rust 翻译，以实现安全性和等价性（2026），https://arxiv.org/pdf/2409 2 53 10506。 v 。Szafraniec, M.、Roziere, B.、Leather, H.、Charton, F.、Labatut, P.、Synnaeve, G.：基于编译器表示的代码转换（2023），https://arxiv.org/abs/ 2207 03578 _ 2207 03578 。，电子印本：。。Wang, B.、Kolluri, A.、Nikolić, I.、Baluta, T.、Saxena, P.：脚本语言的用户可定制转译。ACM 编程语言会议论文集 **7**（OOPSLA1）（2023年4月 3586034 55 10 1145 3586034，10 1145）。https://doi.org/ 。 / https://doi.org/ 。 / 。Wang, B.、Li, T.、Li, R.、Mathur, U.、Saxena, P.：用于自动程序转换的程序骨架。ACM 编程语言会议论文集 **9**（PLDI），920–944（2025）56. Wang, C.、Yu, T.、Xie, C.、Wang, J.、Chen, D.、Zhang, W.、Shi, Y.、Gu, X.、Shen, B.：EVOC2RUST：用于项目级 C 到 Rust 转换的骨架引导框架。arXiv 预印本 arXiv:2508.04295（2025）57. Wang, Y.、Wang, Y.、Wang, S.、Guo, D.、Chen, J.、Grundy, J.、Liu, X.、Ma, Y.、Mao, M.、Zhang, H.、等：RepoTransBench：真实世界的代码仓库级代码翻译基准测试。arXiv 预印本 arXiv:2412.17744（2024）58. xdg-go：GitHub - xdg-go/pbkdf2：在 Go 中实现的 PBKDF2 密钥派生函数（2026 2 59），https://github.com/xdg-go/pbkdf。Xu, Q.、Huang, J.：优化基于大语言模型的 C 到 Rust 翻译中的类型迁移：一种数据流图方法。载于：第 14 届 ACM SIGPLAN 程序分析前沿国际研讨会论文集。第 8–14 页。SOAP '25，ACM（2025）。https://doi.org/10.1145/3735544.3735582，https://doi.org/10 3735582 60 1145 3735544。/。。Yang, A.Z.、Takashima, Y.、Paulsen, B.、Dodds, J.、Kroening, D.：VERT：利用大语言模型进行多语言经验证等价 Rust 转译。载于：自动化软件工程（ASE）（2025）61. Zhan, Q.、Hu, X.、Xia, X.、Li, S.：深度学习算子的经验证提升。arXiv 预印本 arXiv:2412.20992（2024）62. Zhang, H.、David, C.、Wang, M.、Paulsen, B.、Kroening, D.：使用大语言模型对整个项目进行可扩展的验证式代码转换。载于：ACM 编程语言会议论文集。第 9 卷，第 1616–1641 页（2025）。https://doi.org/10.1145/3729315，https://doi.org/10 3729315 63 1145 3729315。/。Zhang, H.、David, C.、Yu, Y.、Wang, M.：所有权引导的 C 到 Rust 转换。载于：计算机辅助验证：第 35 届国际会议，CAV 2023。第 459–482 页。Springer（2023）。https://doi.org/10.1007/978-3-031-37709-9_22，https：//doi.org/ 10 1007 978 3 031 37709 9_22 。/ - - - -。Zhang, J.、Nie, P.、Li, J.J.、Gligoric, M.：使用大语言模型进行多语言代码协同演化。载于：第 31 届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集。第 695–707 页。ESEC/FSE 2023，ACM（2023）。https://doi.org/10.1145/3611643。3616350，3616350 65 10 1145 3611643 https://doi.org/ 。/。。Zhang, Q.、Wang, J.、Xu, G.H.、Kim, M.：HeteroGen：通过自动化测试生成和程序修复将 C 转译为异构 HLS 代码。载于：编程语言与操作系统体系结构支持。第 1017–

1029 页。ASPLOS '22，ACM（2022）。
https://doi.org/10.1145/3503222.3507748，https://doi.org/10 3507748 66 1145
3503222。/。。Zhou, T.、Lin, H.、Jha, S.、Christodorescu, M.、Levchenko, K.、
Chandrasekaran, V.：使用静态分析从 C 到 Rust 的 LLM驱动的多步翻译。arXiv 预印
本 arXiv:2503.12511（2025），https://arxiv.org/pdf/2503.12511

A 一种面向Rust的$ApiIndex$ 构建算法

我们首先形式化有效导入路径与再导出的概念。

路径。 路径p 是用于在crate内引用API项的完全限定名。直观而言，一条路径始于crate根模块，经过零个或多个嵌套子模块，最终终止于项i。然而，并非每一条语法良构路径都是有效导入路径。在Rust中，每个模块都具有相应的可见性，私有模块中包含的条目对外部客户端不可访问。因此，若一条路径经过某个根据Rust隐私规则从crate根不可见的模块，则该路径无效。

定义4（有效导入路径）。 设 m_0 为某个*crate*的根模块。一条语法路径

$$p = m_0 :: m_1 :: \cdots :: m_k :: i$$

是有效导入路径，如果路径上的每个中间模块都公开可见，即

$$\mathsf{Valid}(p) := \forall j \in \{1, \ldots, k\}.\ \mathsf{Public}(m_j).$$

再导出。 Rust 还允许开发者通过再导出，使用其他路径公开条目。重导出语句 `pub use` p 会为 p 引用的条目引入一个别名，使该条目能够通过新路径从重导出模块访问。再导出常用于定义简洁的公共 API，以隐藏内部模块层次；也用于将传递依赖中的特定条目作为 crate 公共接口的一部分公开。

定义 5（重导出解析）。 设 m 为一个模块，p 为一个路径。若 p 解析为条目 i，记作 $p \Downarrow i$，则在m 中的重导出声明 *`pub use`* p 会引入别名路径 $m::\ i'$，使得

$$(p \Downarrow i) \ \Rightarrow\ (m :: i' \rightsquigarrow i).$$

构建 *ApiIndex*。例如，在图 1 中，API `SigningKey`定义于内部模块 `ed25519_dalek::signing`中。由于 `signing`是私有的，因此定义路径对 crate用户而言并非有效导入路径。因此，该 crate 会在根目录再导出 `SigningKey`，使 `ed25519_dalek::SigningKey` 成为有效导入路径。再导出也常用于转发依赖项中的 API。

例如，`Sha512::new`可通过特征 `sha2::Digest`使用；该特征定义于传递依赖项 `digest`中。

要为某个 crate 构建 $ApiIndex$，我们还必须分析其传递依赖，因为 `pub use` 再导出可能会将依赖项定义的 API 作为该 crate 公共接口的一部分公开。如算法 1 所示，我们按后序遍历计算依赖项的 $ApiIndex$，确保在处理当前 crate 之前，这些索引已准备就绪。随后，将这些预计算映射传入目标 crate 的 $ApiIndex$提取过程，并将其作为最终结果返回。

算法 2 详细说明了如何计算每个 crate 的 $ApiIndex$。主入口 ComputeItemsMap 首先调用 traverseModule，构建初始的 $ApiIndex$，其中可能包含方法 API 的占位导入路径。随后，它执行修订阶段，将这些占位符解析为具体导入路径，从而得到完全解析的 $ApiIndex$。

遍历 *crate* 的子程序 递归子程序 traverseModule 会枚举 crate 中所有可公开访问的模块和条目。它会维护当前模块路径前缀 Π，该前缀表示从 crate 根到当前模块的所有外围模块构成的序列，并利用此前缀为发现的每个 API 构造导入路径。

对于形式为 `pub use` $n :: \ldots :: i$ where $n \neq m$ and m 的外部重导出，其中 m 表示当前 crate，根 n 指向传递依赖集合中的一个 crate。在这种情况下，我们从预计算的 $CratesMap$ 中检索 i 的 API 条目，并将其记录在再导出路径 $\Pi' :: i$ 下，以反映它在当前 crate 中公开暴露的位置。

对于形式为 `pub use` $m :: \ldots :: i$ 的内部重导出，如果所引用的路径源自当前 crate 根，我们便将 i 视为在当前模块中本地定义。这样，在遍历过程中，内部重导出就能与模块本地的公共条目统一处理。

最后，对于当前模块中的每个公共条目，以及通过内部重导出引入的条目，我们要么提取其相关文档（针对类型和 trait），要么在该条目本身是子模块时，递归调用 traverseModule 继续遍历。

提取 *API* 的子程序 我们按照算法 3 中概述的方法提取特征、类型和方法。对于方法 API，尽管输入路径 Π 标识了这些方法所属的类型，但无法直接用它来计算这些方法的正确导入路径。这一限制源于 Rust 的 trait 系统：只有在定义该方法的特征 $\mathbb{T}$ 处于作用域内时，才能使用特征方法，因此必须导入 $\mathbb{T}$ 本身。因此，只有在识别出库中的所有特征并确定其导入路径后，才能确定方法 API 的适当导入路径。为处理这一依赖关系，我们首先为方法 API 记录一个占位导入路径，该路径对应于定义该方法的特征 $\mathbb{T}$，并在后续阶段解析。

修订的最后阶段 构建初始的 $ApiIndex$后（其中可能包含方法 API 的占位导入路径），主过程 ComputeIt-

emsMap 会执行最后一个修订阶段。在此阶段，每个占位特征引用都会替换为针对相应特征计算出的具体导入路径，从而为目标库生成完全解析的 $ApiIndex$。

算法 1 主要 API 提取过程

要求： m 根模块
确保： $ApiIndex$公共 API 的文档和可用路径
函数 ExtractCrate(m)
CratesMap $\leftarrow \emptyset$
对于 $n \in$transitiveDependencies(m)，按后序遍历 **执行**
CratesMap $[n] \leftarrow$ ComputeItemsMap($\emptyset$, n, *CratesMap*)
结束循环
返回 *CratesMap* $[m]$
结束函数

路径生成的可靠性。 我们的$ApiIndex$ 构造过程会遍历公共模块树，枚举候选导入路径并收集所有指向可达项的路径，同时还会纳入通过 `pub use`再导出引入的路径。

引理 2（生成导入路径的有效性）。我们构建 *ApiIndex* 时生成的每条路径 p 都满足 Valid(p)。

证明（概要）。路径有两种生成方式。（1）直接路径：从 crate 根开始遍历模块层级，并且只沿着公开可见的子模块前进，即可获得这些路径。因此，按照构造，这类路径上的每个前缀模块都是公开的，所以 Valid(p) 成立。（2）重新导出路径：只有在某个可公开访问的模块中出现 `pub use` 声明时，才会引入重新导出路径。由于重新导出本身是公开的，因此生成的别名路径可从根外部访问，所以 Valid(p) 成立。

B 翻译验证算法

算法 4 总结了翻译过程。

算法 2 计算模块的条目映射

需要：
m 输入模块
$CratesMap$ 将第三方 crate 映射到其条目映射
确保： $ApiIndex$公共 API 集合
函数ComputeItemsMap(m, $CratesMap$)
    ApiIndex $\leftarrow$ traverseModule($\emptyset$, m, ***CratesMap***)
    遍历 $f \in$ 函数(***ApiIndex***) **执行**
        文档 $\mathbb{T} \leftarrow$ ***ApiIndex***$[f]$
        如果 $\mathbb{T} \in$ ***trait***(***ApiIndex***) **则**
            $_\, p \leftarrow$ ***ApiIndex***$[\mathbb{T}]$
            ApiIndex$[f] \leftarrow$ 文档 p
        else
            删除（***ApiIndex***, f）
        结束条件
    结束循环
    返回*ApiIndex*
结束函数
函数traverseModule(Π, m, $CratesMap$)
    $\Pi' \leftarrow \Pi :: m$
    ApiIndex $\leftarrow \emptyset$
    遍历 `pub use` $n :: \ldots :: i \in$ 外部重导出(m) **执行**
        Doc, $_ \leftarrow$ ***CratesMap***$[n][i]$
        ApiIndex$[i] \leftarrow$ 文档 $\Pi' :: i$
    结束循环
    内部重新导出项 $\leftarrow \emptyset$
    对于 `pub use` $m :: \ldots :: i \in$ 内部重新导出(m) **执行**
        追加(内部重新导出项, i)
    结束循环
    对于 $i \in$ 公共项(m) $\cup$内部重新导出项 **执行**
        如果 i 是模块**则**
            ApiIndex $\leftarrow$***ApiIndex*** $\cup$ traverseModule(Π', i, ***CratesMap***)
        否则，如果 i 是特征，**则**
            ApiIndex $\leftarrow$***ApiIndex*** $\cup$ 提取特征(Π', i)
        否则，如果 i 是类型，**则**
            ApiIndex $\leftarrow$***ApiIndex*** $\cup$ 提取类型(Π', i)
        结束条件判断
    结束循环
    返回 ***ApiIndex***
函数结束

算法 3 提取函数、类型和 trait

函数 extractType(Π, t)
  ApiIndex $\leftarrow \{t \mapsto (\text{文档}(t),\ \Pi ::\ t)\}$
  针对 $(\text{实现}\ \ \mathbb{T}\ \text{针对}\ t\{f\}) \in \text{实现块}(t)$ **执行**
    针对 $f \in f$ **执行**
      ApiIndex$[f] \leftarrow \text{文档}\ (f)\ \ \mathbb{T}$
    结束 for
  结束 for
结束函数
函数 extractTrait(Π, $\mathbb{T}$)
  返回 $\{\mathbb{T} \mapsto (\text{文档}(\mathbb{T})\ \ \Pi ::\ \mathbb{T})\}$
函数结束

算法 4 建立跨语言互操作性

要求： T_{Go}，T_{Rust}，以及Go 值 v $^{-}$
确保： T_{Proto}，$\mathcal{A}_{T\,\text{Lang}}$ 以及 $\mathcal{A}^{-}_{T_{\text{语言}}}$ 其中 语言 $\in \{\text{Go}, \text{Rust}\}$
函数GenInteroperators(T_{Go}，T_{Rust})
  /*生成 Proto 模式、Go 粘合代码，并验证 Go $\leftrightarrow$ Proto*/
  当 *true* **做**
    $T_{\text{Proto}} \leftarrow$生成 Proto 模式($T_{\text{Go}}$)
    $\mathcal{A}_{T_{\text{Go}}}$，$\mathcal{A}^{-}_{T_{\text{Go}}} \leftarrow$ GenHighLevelGlueCode(T_{Go}，T_{Proto})
    如果CheckRoundtrip(T_{Go}，T_{Proto}，$\mathcal{A}_{T\ \text{Go}}$ $\mathcal{A}^{-}_{T_{\text{Go}}}$ v) **则**
      跳出
    结束如果
  结束当
  /*生成 Rust 胶水代码并验证 Go $\leftrightarrow$ Rust*/
  当 *true* **时**
    $\mathcal{A}_{T_{\text{Rust}}}$ $\mathcal{A}^{-}_{T}$ $\mathcal{A}_{T\ \text{Rust}}$ $\mathcal{A}^{-}_{T\ \text{Go}} \leftarrow$ GenHighLevelGlueCode(T_{Rust}，T_{Proto})
    $\mathcal{A}^{-}_{T}$，$\mathcal{A}_{T\ \text{Rust}} \circ \mathcal{A}_{T_{\text{Rust}}}$，$\mathcal{A}_{T\ \text{Rust}} \circ \mathcal{A}_{T_{\text{Go}}}$
    如果 CheckRoundtrip(T_{Go}，T_{Rust}，$\mathcal{A}_{T}$，$\mathcal{A}^{-}_{T}$，v) **则**
      break
    结束 if
  结束 while
结束函数