APRESENTAÇÃO

HackAtom

VoiD

BackStage

Bruno C. Barreto
Fullstack Head Developer.

Thiago Gouveia
Economist, specialist in
Blockchain and cryptocurrencies

Levi Roriz
Project Manager
and Creative Editor

Lucas Emanuel
Fullstack Developer

Daniel Affonso
Fullstack Developer

VoiD

Documento Descritivo do Projeto

Transpilador cross-chain para smart-contracts na WEbdEX

1. Visão Geral do Projeto

1.1. O Que é o Projeto?

Este projeto é uma ferramenta transpiler projetada para facilitar a migração de smart contracts da blockchain Polygon (escritos em Solidity) para a blockchain Solana (escritos em Rust). O projeto é desenvolvido em Rust e utiliza uma abordagem de Linguagem Específica de Domínio (DSL) para garantir um processo de conversão preciso e simplificado. Inicialmente, o foco será converter contratos Solidity para Rust para o evento HackAtom, com planos de estender a funcionalidade após o evento para suportar a tradução bidirecional por meio de uma DSL intuitiva e interface guiada para o usuário.

1.2. O Que o Projeto Faz?

- Processo de Transpiração:
    - Parsing: Realiza o parsing do código-fonte em Solidity para uma Árvore de Sintaxe Abstrata (AST) utilizando bibliotecas em Rust, como *pest.rs* e *nom*.
    - Representação Intermediária (IR): Converte a AST em uma Representação Intermediária (IR) que padroniza a semântica central do contrato.
    - Reconstrução: Transforma a IR em uma nova AST projetada para a linguagem alvo (Rust), e então gera código equivalente em Rust que adere aos paradigmas de smart contracts da Solana.
- DSL Planejada e Interface de Usuário:
    - Após o HackAtom, a solução incluirá uma DSL para criação de contratos, permitindo que os desenvolvedores escrevam smart contracts de forma agnóstica à plataforma.
    - Uma interface intuitiva, aprimorada por um agente guiado por IA, auxiliará os usuários na criação e migração dos contratos.

2. Necessidade e Motivação

2.1. Por Que o Projeto é Necessário

- Interoperabilidade:
  A interoperabilidade entre cadeias é crucial para a evolução das finanças descentralizadas (DeFi). Migrar contratos entre blockchains, como Polygon e Solana, pode abrir novos mercados, melhorar a liquidez e oferecer flexibilidade tanto para desenvolvedores quanto para usuários.
- Eficiência no Desenvolvimento:
  Reescrever smart contracts manualmente ao mudar de ambiente blockchain é propenso a erros e consome muito tempo. Um transpiler automatizado minimiza erros humanos e acelera o processo de migração.
- Extensibilidade e Preparação para o Futuro:
  A DSL planejada fornece uma maneira unificada de escrever smart contracts, garantindo que os desenvolvedores possam direcionar múltiplas cadeias sem a necessidade de dominar diferentes linguagens e paradigmas de programação.

3. Características do Projeto

3.1. Características Funcionais

- Conversão de Fonte para Destino:
  A função principal é converter com precisão contratos baseados em Solidity para código em Rust compatível com o modelo de smart contracts da Solana.
- Transformação de AST e IR:
    - Parsing eficiente de contratos Solidity para uma AST.
    - Conversão da AST para uma Representação Intermediária (IR) que encapsula a lógica do contrato.
    - Reconstrução da IR em uma AST específica para Rust, que é então serializada em código Rust válido.
- Design Extensível:
  Projetado com modularidade em mente, de modo que linguagens ou funcionalidades adicionais (por exemplo, a DSL) possam ser integradas após o HackAtom.

3.2. Características Não-Funcionais

- Desempenho:
  O transpiler deve ser eficiente ao processar contratos grandes e complexos, utilizando execução assíncrona via runtime Tokio.
- Manutenibilidade:
  O código deve ser limpo, modular e bem documentado. O uso de bibliotecas de parsing consolidadas garante que cada componente seja testável e de fácil manutenção.
- Experiência do Usuário (Componente Futuro):
  Após o evento, o projeto incluirá uma interface guiada por IA, garantindo que mesmo usuários com conhecimento limitado em blockchain ou programação possam migrar e criar smart contracts com sucesso.

- Escalabilidade:
  O sistema é projetado para lidar com bases de código em crescimento e com alvos de linguagem adicionais em iterações futuras.

4. Requisitos Funcionais

1. Entrada de Código Solidity:
   - Aceitar arquivos de código-fonte em Solidity como entrada.
   - Validar a sintaxe e a semântica antes do processamento.
2. Geração da AST:
   - Utilizar as bibliotecas *pest.rs* e *nom* para fazer o parsing do código-fonte em Solidity.
   - Gerar uma AST que represente com precisão a estrutura do código.
3. Criação da Representação Intermediária (IR):
   - Converter a AST de Solidity em uma IR que capture a lógica essencial, o estado e as funções.
   - Garantir que a IR seja genérica o suficiente para suportar a conversão para múltiplas linguagens alvo.
4. Geração da AST em Rust e Emissão de Código:
   - Transformar a IR em uma AST compatível com Rust.
   - Gerar o código final em Rust que esteja em conformidade com o framework de smart contracts da Solana.
5. Tratamento de Erros:
   - Fornecer mensagens de erro descritivas para erros de sintaxe, problemas de transformação ou construções não suportadas.
6. Integração Futura da DSL (Pós-HackAtom):
   - Projetar uma DSL abstrata que suporte a geração de código tanto para Solidity quanto para Rust.
   - Desenvolver uma interface com front-end acompanhada por um agente guiado por IA para auxiliar os usuários na criação e migração dos contratos.

5. Requisitos Não-Funcionais

1. Desempenho e Escalabilidade:
   - Otimizar o parsing e a geração de código para lidar eficientemente com contratos de grande porte.
   - Utilizar operações assíncronas via runtime Tokio para garantir responsividade.
2. Modularidade e Manutenibilidade:
   - O código deve ser bem organizado, com limites claros entre módulos para parsing, transformação de IR e geração de código.
   - Incluir documentação inline extensa e documentação externa para desenvolvedores.
3. Robustez e Tratamento de Erros:
   - Implementar mecanismos abrangentes de detecção e recuperação de erros.
   - Garantir que as mensagens de erro sejam claras e orientem os desenvolvedores na resolução dos problemas.
4. Segurança:
   - Validar todas as entradas para evitar injeção de código ou outras explorações maliciosas.
   - Seguir as melhores práticas de codificação segura em Rust.
5. Experiência do Usuário (para a Interface da DSL):
   - Projetar uma interface intuitiva e amigável.
   - Integrar um agente de IA que possa fornecer orientações em tempo real e solucionar problemas.

6. Como Funciona

6.1. Fluxo do Sistema

1. Etapa de Entrada:
   - Os desenvolvedores fornecem os smart contracts em Solidity como arquivos de entrada.
   - O sistema valida a entrada quanto à sintaxe correta e aos construtos esperados.
2. Etapa de Parsing:
   - O código em Solidity é convertido em uma AST utilizando *pest.rs* ou *nom*.
   - Erros detectados durante esta fase são reportados ao desenvolvedor.
3. Etapa de Transformação da IR:
   - A AST gerada é transformada em uma Representação Intermediária (IR).
   - A IR abstrai a lógica e a estrutura chave, atuando como uma ponte entre as duas linguagens.

4. Etapa de Geração de Código:
    - A IR é utilizada para gerar uma nova AST que representa a linguagem alvo (Rust).
    - Por fim, a AST em Rust é serializada em código-fonte que atende aos requisitos de smart contracts da Solana.
5. Etapa de Saída:
    - O código em Rust gerado é disponibilizado para o desenvolvedor.
    - Logs detalhados e relatórios de erros são fornecidos caso ocorram problemas.
6. Aprimoramentos Futuros:
    - Após o HackAtom, o projeto será expandido para incluir uma DSL e uma interface amigável com um agente de IA para orientar tanto na criação quanto na migração dos contratos.

7. Tecnologias Utilizadas e Justificativas

7.1. Linguagem de Programação

- Rust:
    - Escolhida por seu desempenho, segurança e suporte à concorrência.
    - Garante o manuseio seguro e eficiente da transpiração de smart contracts.

7.2. Bibliotecas de Parsing

- pest.rs e nom:
    - Utilizadas para realizar o parsing robusto e a geração da AST.
    - Suas capacidades de parsing combinatório as tornam ideais para lidar com as complexidades da sintaxe do Solidity.

7.3. Runtime Assíncrono

- Tokio:
    - Utilizado para lidar eficientemente com operações assíncronas.
    - Melhora o desempenho, especialmente ao processar contratos grandes ou múltiplos arquivos simultaneamente.

7.4. Bibliotecas para Manipulação de Strings

- Bibliotecas adicionais serão utilizadas para manipulação e concatenação de strings, garantindo que a transformação da IR para código seja realizada de forma eficiente e sem erros.

7.5. Gerador de Parser (Opcional para Solidity)

- ANTLR:
    - Pode ser utilizado como uma ferramenta alternativa ou complementar para gerar o parser do Solidity.
    - Oferece flexibilidade e confiabilidade comprovada para parsing de gramáticas complexas.

8. Roteiro do Projeto e Aprimoramentos Futuros

8.1. Metas Imediatas (HackAtom)

- Desenvolver o transpiler central que converte smart contracts em Solidity para Rust.
- Focar na geração robusta da AST e da IR, garantindo uma transformação de código sem erros.
- Validar o desempenho utilizando um conjunto de exemplos de contratos do mundo real.

8.2. Aprimoramentos Pós-HackAtom

- Desenvolvimento da DSL:
    - Implementar uma linguagem específica de domínio para escrever smart contracts.
    - Garantir que a DSL seja abstrata o suficiente para ser compilada tanto para Solidity quanto para Rust.
- Interface de Usuário & Integração com IA:
    - Desenvolver uma interface front-end intuitiva que guie os usuários durante a migração dos contratos. ○ Integrar um agente de IA que ofereça assistência contextual e resolução de problemas.
- Suporte a Linguagens Estendidas:
    - Explorar a possibilidade de adicionar suporte para plataformas blockchain ou linguagens adicionais.

9. Conclusão

Este projeto visa simplificar o processo de migração de smart contracts entre blockchains, automatizando a conversão de Solidity (Polygon) para Rust (Solana). Ao adotar uma abordagem modular com parsing robusto, transformação de IR e geração de código, a ferramenta minimiza erros humanos e acelera o processo de migração. Os aprimoramentos futuros, incluindo a DSL e uma interface guiada por IA, prometem tornar o desenvolvimento de smart contracts mais acessível e eficiente para desenvolvedores em todo o ecossistema DeFi.

Este documento descritivo serve como um guia completo dos objetivos, do design e da arquitetura técnica do projeto, permitindo que os desenvolvedores compreendam plenamente e reproduzam a ferramenta de migração de smart contracts, garantindo clareza tanto na implementação quanto na escalabilidade futura da solução.