Arquitectura y Viabilidad del Optimizador Matemático Automático en Lean 4

El diseño de un Optimizador Matemático Automático en Lean 4 para código Rust, centrado en la viabilidad técnica y la garantía formal, se basa en la integración de la metaprogramación avanzada de Lean 4, la vasta biblioteca Mathlib, y principios de optimización verificada. El proyecto es viable y prometedor, utilizando un enfoque de "correcto por construcción" para generar código Rust optimizado con garantías semánticas.

1. Input Translation (Rust to Hacspec)

Rol de Hacspec:

Hacspec es un subconjunto de Rust puramente funcional, diseñado específicamente para la especificación formal de algoritmos criptográficos. Su principal ventaja radica en la restricción de características de Rust que complican la verificación formal, tales como el borrowing mutable, efectos secundarios, punteros y unsafe code. Al limitar el lenguaje a expresiones funcionales y tipos de datos inmutables, Hacspec simplifica enormemente el modelado semántico del código, haciéndolo amenable a herramientas de verificación. Permite modelar restricciones de dominio utilizando el sistema de tipos de Rust.

Proceso de Traducción:

La traducción de Rust a Hacspec implica un frontend que parsea el código Rust de entrada y verifica que cumple con las restricciones de Hacspec. Esto puede implementarse como un linter o un pasador de análisis estático que, al éxito, genera una Representación Intermedia (IR) o un Árbol de Sintaxis Abstracta (AST) que se adhiere a la especificación Hacspec. Esta IR sería el punto de partida para el levantamiento a Lean 4.

Ejemplo de Hacspec:

```
// Hacspec: Suma modular de enteros sin signo
type U32 = hacspec_lib::U32;

pub fn add_mod(a: U32, b: U32) -> U32 {
    a + b
}

pub fn mul_mod(a: U32, b: U32) -> U32 {
    a * b
}
```

2. Lifting to Lean 4

El proceso de "levantamiento" (lifting) es la transformación de la representación Hacspec (AST o IR) en tipos inductivos de Lean 4 o directamente en Expr de Lean. Esto establece el puente entre el código fuente y el entorno de prueba de teoremas.

Proceso Arquitectónico:

Definición de Tipos Inductivos en Lean: Se definen tipos inductivos en Lean 4 que modelan la estructura del AST de Hacspec. Por ejemplo,

para expresiones aritméticas básicas:

```
inductive HacspecExpr where
  | const : Nat → HacspecExpr
  | var   : String → HacspecExpr
  | add   : HacspecExpr → HacspecExpr → HacspecExpr
  | mul   : HacspecExpr → HacspecExpr → HacspecExpr
  | app   : HacspecExpr → HacspecExpr → HacspecExpr
  | lam   : String → HacspecExpr → HacspecExpr
  -- ... otros constructores para tipos, funciones, etc.
```

Mapeo de Tipos: Los tipos primitivos de Hacspec (ej., U32) se mapean a tipos formales en Lean 4 (ej., Fin (2^32) para enteros de 32 bits, o Nat con un módulo asociado). Esto es crucial para aplicar teoremas de Mathlib.

Conversión a Lean.Expr: Si bien los tipos inductivos son útiles para la representación inicial, el motor de optimización a menudo operará directamente sobre la estructura Lean.Expr nativa, que es el tipo fundamental para términos y tipos en el kernel de Lean. Un elab_meta personalizado o una función de traducción convertiría la HacspecExpr en Lean.Expr.

Manejo de Variables Ligadas (De Bruijn): Al convertir a Lean.Expr, es esencial utilizar la representación de variables de De Bruijn para manejar correctamente las variables ligadas (Expr.bvar), evitando colisiones de nombres y simplificando la manipulación de términos en el optimizador.

Ejemplo de Levantamiento (conceptual):

Un término Hacspec como a + b podría levantarse a un Lean.Expr que representa HAdd.hAdd a b donde a y b son variables con sus tipos inferidos.

```
-- Ejemplo de función de lifting (simplificada)
def liftHacspecExprToLeanExpr (e : HacspecExpr) : MetaM Expr := do
  match e with
  | .const n => pure $ mkRawNatLit n
  | .var s   => pure $ mkConst (Name.mkSimple s)
  | .add e1 e2 => do
    let leanE1 ← liftHacspecExprToLeanExpr e1
    let leanE2 ← liftHacspecExprToLeanExpr e2
    let add_fn ← mkConst `HAdd.hAdd
    pure $ mkAppN add_fn #[leanE1, leanE2] -- Simplificado: requiere inferencia de tipo real
  -- ... y así sucesivamente para otros constructores
```

3. Optimization Engine (Equality Saturation)

El corazón del optimizador será un motor basado en Equality Saturation y E-graphs, implementado o adaptado dentro del entorno de Lean 4.

Fundamentos Teóricos:

E-graphs: Son estructuras de datos que representan conjuntos de términos equivalentes. Cada nodo en un E-graph, llamado E-class, almacena un conjunto de E-nodes que son sintácticamente diferentes pero semánticamente equivalentes. Un E-node es una expresión con sus argumentos reemplazados por los E-classes a los que pertenecen.

Congruence Closure: Cuando se descubre una nueva igualdad entre dos

términos, el algoritmo de congruence closure propaga esta igualdad a través del E-graph, fusionando E-classes si sus E-nodes correspondientes se vuelven equivalentes.
Equality Saturation: Es un proceso iterativo en el que se aplican reglas de reescritura a un E-graph. Cada aplicación de regla puede introducir nuevas equivalencias, lo que lleva a la fusión de E-classes y la expansión del E-graph. Este proceso continúa hasta que no se pueden encontrar nuevas equivalencias (saturación).
Función de Costo: Una vez saturado el E-graph, se utiliza una función de costo heurística para seleccionar el término "óptimo" de cada E-class. Esta función puede considerar factores como el tamaño del término, la complejidad de las operaciones, el uso de registros, etc.
Implementación en Lean 4:
La implementación se centrará en manipular Lean.Expr dentro de la mónada MetaM:

Representación de E-graphs: Se definirían tipos inductivos o estructuras Lean para modelar EClass y ENode, donde los E-nodes contendrían Lean.Expr (o identificadores que apuntan a Lean.Exprs almacenados en un RBMap/HashMap).

```
structure ENode (α : Type) where
  op    : Name
  args  : Array α -- indices to other EClasses
  -- ... other metadata

structure EClass (α : Type) where
  id      : Nat
  nodes   : Array (ENode α)
  members : Array Expr -- actual Lean expressions
  -- ... other metadata (cost, parents, etc.)

abbrev EGraph := Std.HashMap Nat (EClass Nat) -- Mapping EClass ID to EClass
```

Operaciones sobre E-graphs: Las funciones para insert, union (fusionar E-classes), rebuild (aplicar congruence closure), y apply_rules (aplicar reglas de reescritura) se escribirían en MetaM.
Integración con MetaM: MetaM es fundamental para:
Acceso al LocalContext: Para resolver nombres de variables y tipos.
reduce y isDefEq: Utilizar las capacidades de reducción del kernel de Lean y la verificación de igualdad definicional para canonizar términos y validar equivalencias. Esto es crucial para que las reglas de reescritura sean semánticamente correctas.
mkApp / mkLambda: Construir nuevos términos Lean.Expr a partir de sub-expresiones optimizadas.
Alternativas y Consideraciones: Si bien la re-implementación completa de egg en Lean sería un esfuerzo considerable, la experiencia de egg informaría la arquitectura. Una implementación nativa en Lean 4 se beneficiaría directamente del kernel verificado y de la infraestructura de MetaM, lo que garantizaría la corrección. El rendimiento sería una consideración clave, y se podrían usar estructuras de datos eficientes de Std (como HashMap, RBMap, PersistentArray).
4. Mathematical Guidance (Mathlib Integration)

Mathlib es el pilar para la corrección formal de las optimizaciones.

Aprovechamiento de Mathlib:

Base de Teoremas Verificados: Mathlib proporciona una vasta colección de matemáticas formalizadas, desde estructuras algebraicas básicas (semigrupos, monoides, grupos, anillos, cuerpos) hasta álgebra lineal y análisis.
Reglas de Reescritura: Cada teorema de igualdad (Eq) o equivalencia lógica (Iff) en Mathlib puede ser considerado una regla de reescritura. El optimizador identificará patrones en el Lean.Expr que coincidan con el lado izquierdo de una igualdad y los reescribirá con el lado derecho, utilizando la potencia de MetaM.isDefEq para verificar la aplicabilidad.
Ejemplos de Reglas:
Asociatividad: (a + b) + c = a + (b + c) (add_assoc).
Conmutatividad: a ∗ b = b ∗ a (mul_comm).
Distributividad: a ∗ (b + c) = a ∗ b + a ∗ c (mul_add).
Reducción Modular: (a % N + b % N) % N = (a + b) % N.
Optimización de Cuerpos Finitos: Identificación de patrones para aplicar algoritmos más eficientes en dominios específicos (ej., exponenciación rápida modular, optimizaciones de curvas elípticas como las presentes en el trabajo de Fiat-Crypto).
Garantía de Semántica: Dado que los teoremas de Mathlib están formalmente demostrados, cualquier transformación aplicada a través de estas reglas de reescritura garantiza la preservación de la semántica denotacional del programa original. Esto es fundamental para la "verificación por construcción".
Reconocimiento de Estructuras Algebraicas: El optimizador necesitará una capacidad para reconocer cuándo una expresión Lean.Expr opera dentro de una instancia de una estructura algebraica (ej., un Ring o Field). Esto puede lograrse mediante inferencia de tipos y consulta de instancias en el LocalContext de MetaM.
5. Lean 4 Architecture
La arquitectura interna de Lean 4 es el entorno operativo para el optimizador.

Metaprogramación:

MetaM: Es la mónada central para el optimizador. Permite interactuar con el environment de Lean (donde se almacenan las definiciones globales), el local context (variables locales y sus tipos), y el kernel (para verificación de tipos y reducción). MetaM provee las herramientas para:
Manipular Expr (términos abstractos).
Realizar reduceExpr (reducción de expresiones según las definiciones).
Comprobar isDefEq (igualdad definicional).
Instanciar metavariables.
CoreM, TermElabM, TacticM:
CoreM: Proporciona acceso al estado más básico del kernel.
TermElabM: Utilizado en el frontend de Lean para elaborar Syntax en Expr tipados, manejando inferencia de tipos implícitos y metavariables. El proceso de "lifting" a Lean.Expr podría usar o ser

influenciado por TermElabM.
TacticM: Mónada para la implementación de tácticas de prueba. Aunque el optimizador principal no sería una "táctica" en el sentido tradicional, ciertas estrategias de búsqueda o aplicación de reglas podrían inspirarse en el diseño de tácticas.
Expr y Syntax: El código Rust inicial se convierte en Syntax y luego se elabora en Expr. El motor de optimización operará principalmente sobre Expr para aprovechar el kernel de Lean y MetaM. Syntax sería el objetivo de la etapa final de generación de código.
Reflexión: La capacidad de Lean para definir sintaxis personalizada y rutinas de elaboración (elab macros) es vital para construir un frontend robusto que pueda interpretar y representar el código Hacspec de manera idiomática en Lean, o incluso definir un DSL incrustado para Hacspec si fuera necesario.
TransparencyMode: Controla la visibilidad de las definiciones durante la reducción. Es crucial para el optimizador elegir el modo de transparencia adecuado (ej., reducible, instance, all) para aplicar las reglas de reescritura correctamente, permitiendo el despliegue de constantes cuando sea beneficioso para la optimización, o abstrayéndolas para análisis de alto nivel.
6. Rust Verification and Code Generation
La etapa final cierra el ciclo, traduciendo la Lean.Expr optimizada de nuevo a código Rust, manteniendo la cadena de verificación.

Extracción del E-graph Óptimo:

Selección del Término: Después de que el Equality Saturation ha completado su ejecución y el E-graph ha sido saturado, se utiliza la función de costo para recorrer el E-graph y extraer el Lean.Expr que representa la versión más eficiente del programa.
Canonización: El término extraído estará en una forma canónica dentro de Lean.Expr gracias a las reducciones y equivalencias aplicadas.
Traducción Inversa (Lean Expr a Hacspec/Rust AST):

Lean.Expr a Hacspec AST: Este es el paso inverso al "lifting". Una función de des-elaboración o una función recursiva recorrerá el Lean.Expr optimizado y reconstruirá el AST de Hacspec correspondiente. Se deberá manejar la conversión de tipos Lean (ej., Fin (2^32)) de nuevo a tipos Hacspec (ej., U32).
Manejo de De Bruijn: Las variables de De Bruijn en Lean.Expr deben convertirse de nuevo a nombres de variables legibles para Rust, asegurando que no haya colisiones y que el alcance sea correcto.
Generación de Código Rust:

Hacspec AST a Código Rust: Una vez que se tiene el AST optimizado en formato Hacspec, un generador de código final lo serializa a código fuente Rust. Este generador debe adherirse estrictamente a la sintaxis y las convenciones de Hacspec.
Cadena de Verificación:
La principal fortaleza de este enfoque es la "corrección por construcción":

Especificación Verificable: Hacspec, por su diseño restringido,

facilita el modelado formal.
Lifting Semánticamente Preservante: La traducción a Lean.Expr debe ser verificada para preservar la semántica del programa Hacspec.
Optimización Formalmente Verificada: Todas las transformaciones aplicadas por el motor de Equality Saturation se basan en teoremas de Mathlib probados en Lean, lo que garantiza que las optimizaciones preservan la semántica.
Generación Correcta: La fase de des-elaboración y generación de código debe ser demostrada como una traducción semánticamente equivalente de la Lean.Expr optimizada al código Rust.
Este enfoque elimina la necesidad de verificación a posteriori del ejecutable generado, ya que la corrección se mantiene a lo largo de todo el pipeline, desde la especificación hasta el código final optimizado. El resultado es un compilador que no solo optimiza el código, sino que también lo hace con una garantía formal de corrección semántica, una característica que los compiladores tradicionales como LLVM no pueden ofrecer sin esfuerzos de verificación adicionales.

Ejemplo de Flujo General de Archivos (Pseudocódigo):

```
/
├── hacspec_frontend/
│   ├── src/
│   │   ├── parser.rs         // Parses Rust code into Hacspec AST
│   │   ├── checker.rs        // Verifies Hacspec compliance
│   │   └── ast.rs            // Defines Hacspec AST structures
│   └── Cargo.toml
│
├── lean_optimizer/
│   ├── Leanpkg.toml
│   ├── Lakefile.lean
│   └── HacspecOptimizer/
│       ├── Main.lean
│       ├── Core.lean         // Definitions for HacspecExpr, Type
mapping
│       ├── Lifting.lean      // Hacspec AST to Lean Expr
│       ├── EGraph.lean       // E-graph data structures and
operations (MetaM)
│       ├── Rules.lean        // Integration of Mathlib theorems as
rewrite rules
│       ├── Optimizer.lean    // Equality Saturation control logic
│       ├── Lowering.lean     // Lean Expr to Hacspec AST
(optimized)
│       └── MathlibExtensions/  // Specific Mathlib lemmas for
algebraic structures
│           ├── RingOps.lean
│           └── FieldOps.lean
│
└── code_generator/
    ├── src/
    │   ├── ast_to_rust.rs    // Converts Hacspec AST back to Rust
code string
    │   └── main.rs           // CLI for code generation
```

```
└── Cargo.toml
```