Enhancing the Interface Signature Graph: A Roadmap to a Comprehensive Semantic Model

Preamble: The Evolution from Signature Graph to Semantic Model

The existing Interface Signature Graph (ISG), with its representation of Calls, Implements, and Uses relationships, provides a strong syntactic foundation for code analysis. The proposed enrichments—Interface Context, Control-Flow Fingerprints, Data-Flow Stubs, and Effect Summaries—are logical next steps toward capturing more nuanced program behavior. However, to achieve a level of reasoning that rivals modern IDEs and compilers, a more profound architectural evolution is required.

This report outlines a roadmap to transform the ISG from a graph of syntactic declarations into a multi-layered, queryable semantic model. The architectural north star for this evolution is rust-analyzer, a system that demonstrates a robust, scalable approach to static analysis through a layered design.[1] It separates concerns into distinct intermediate representations (IRs): a syntax layer (syntax), a definition and name resolution layer (hir-def), a type and trait system layer (hir-ty), and an IDE-feature layer (ide).[2] By mirroring this layered philosophy, the ISG can evolve to manage complexity and meet stringent performance contracts, such as those defined for the parseltongue project.

This evolution involves three major stages of enrichment:

1. **Deep Semantic and Structural Enrichment:** Building a High-Level Intermediate Representation (HIR) analogue to capture the full declarative semantics of the code.
2. **Deep Type System and Trait Integration:** Incorporating a comprehensive type inference and trait resolution engine, mirroring a compiler's type-checking phase.
3. **Granular Flow Analysis:** Lowering function bodies into a Mid-level Intermediate Representation (MIR) analogue to enable path-sensitive control and data-flow analysis.

Section 1: Deep Semantic and Structural Enrichment (The HIR-def Analogue)

This phase expands upon the "Interface Context" concept by modeling fundamental language constructs as structured, queryable data rather than simple annotations. This approach is directly inspired by the capabilities of rust-analyzer's hir-def and hir crates, which form the bedrock of its semantic understanding.[2]

1.1 Structured Visibility Modeling

To reason effectively about API surfaces and encapsulation, visibility must be treated as a first-class citizen. The rust-analyzer hir crate documentation lists a Visibility enum for this purpose.[3] While its precise definition is not available in the provided materials, the syn crate—already a dependency of parseltongue —provides an excellent structural model with variants like Public, Restricted, and Inherited.[4]

The visibility property on an ISG node should be elevated from a simple boolean or string to a structured enum:

Rust

```
pub enum Visibility {
    Public,
    Crate, // pub(crate)
    Path(hir::Path), // pub(in path)
    Private,
}
```

This structured representation enables precise, scope-aware queries that are impossible with simple flags. It becomes trivial to ask questions such as, "Find all public API functions exposed by this module that are visible to a downstream crate," or "Identify all items that are pub(crate) but are only used within their parent module, making them candidates for

privatization."

1.2 Comprehensive Attribute Representation and Querying

Attributes (#[...]) are a core part of Rust's metadata system, conveying everything from conditional compilation and testing to documentation and deprecation notices. rust-analyzer's hir crate features a sophisticated system for this, including Attrs, Attr, and the HasAttrs trait.[3]

Instead of treating attributes as unparsed text, NodeData should be augmented with a Vec<Attribute>, where Attribute is a structured type capable of representing the full spectrum of attribute forms:

- **Word:** #[test]
- **List:** #[cfg(any(unix, windows))]
- **Name-Value:** #[deprecated(since = "1.2.0", note = "Use new_func instead")]

This allows the ISG to become a direct source for critical metadata, enabling powerful queries:

- "Find all functions marked #[deprecated] and extract their note."
- "Extract and render all documentation comments for this struct and its fields."
- "Identify all functions that are part of the test suite (#[test]) and are not marked #[ignore]."

1.3 Modeling Conditional Compilation (cfg) Graphs

Rust code is not monolithic; its very structure is conditional. rust-analyzer addresses this by having a dedicated cfg crate to parse and evaluate conditional compilation expressions.[2] A static analysis tool that ignores cfg attributes is analyzing a program that may not actually exist in any real-world compilation.

The most significant architectural enhancement in this phase is to make the ISG cfg-aware. This requires two key changes:

1. **Attach CfgExpr to Graph Elements:** Every NodeData and EdgeKind that can be conditionally compiled must have an associated Option<CfgExpr>. CfgExpr would be a structured representation of the #[cfg(...)] attribute's logic (e.g., an expression tree for any(unix, all(target_arch = "wasm32", feature = "js"))).
2. **Introduce a "Configuration Context" for Queries:** Queries must be executed against

the ISG *given* a specific configuration (e.g., target=windows, features=["serde"]). The query engine would then traverse only the nodes and edges whose CfgExpr evaluates to true within that context.

This transforms the ISG from a single, static graph into a powerful template for generating context-specific graphs. This paradigm shift enables reasoning about the code's behavior across its entire configuration space, answering questions like:

- "What is the call graph for this function when compiled for wasm32?"
- "Which structs are completely removed when the serde feature is disabled?"
- "Show the API surface available only on nightly Rust."

This approach moves beyond simply adding metadata to the graph; it fundamentally redefines the graph as a conditional structure, reflecting the reality of how complex Rust projects are built and behave.

Section 2: Deep Type System and Trait Integration (The HIR-ty Analogue)

Reasoning about Rust is fundamentally reasoning about its type system. This section proposes evolving the ISG to incorporate a rich model of types, generics, and traits, mirroring the capabilities of rust-analyzer's hir-ty crate.[2] This is essential for understanding polymorphism, resolving method calls, and analyzing the true semantics of generic code.

2.1 Full Type Inference and Generic Substitutions

A function's signature provides only a partial view of its types. The concrete types of local variables, intermediate expressions, and generic parameters are determined through type inference. rust-analyzer's hir-ty crate performs this deep inference, storing the resolved type for every expression and pattern in the program.[2]

To achieve this level of understanding, the ISG must be enriched as follows:

1. **Introduce a hir::Ty Representation:** A robust data structure, analogous to rustc's internal Ty, must be created. This structure needs to represent not just named types but also primitives, references (&T), function pointers (fn(i32) -> bool), generic parameters (T), inference variables (_), and trait objects (dyn Trait).

2. **Store Per-Expression Inferred Types:** For each function node, the ISG should store an InferenceResult map, which links every expression within the function's body to its fully inferred and canonicalized hir::Ty.
3. **Capture Generic Substitutions on Edges:** The Calls edge is incomplete without knowing *how* a generic function is being called. This edge must be augmented to store the GenericArgs (the list of types and constants used to substitute the generic parameters) for that specific call site. For example, a call to Vec::push(5) would have a Calls edge annotated with a substitution mapping T to i32.

This enrichment enables a new class of precise semantic queries:

- "What is the concrete type of the variable x at this line, even if its declared type is impl Trait?"
- "Show all call sites of Vec::push where T is instantiated as String."
- "Find all expressions that result in a type mismatch," a diagnostic directly supported by rust-analyzer.[2]

2.2 Advanced Trait Resolution Model

Rust's power lies in its trait system. A simple Implements edge is insufficient to model this. rust-analyzer's hir-ty crate contains a sophisticated trait solver (next_solver) that handles trait bounds, associated types, and method resolution.[2]

To model traits correctly, the ISG should be enhanced:

1. **Model impl Blocks:** The Implements edge should be replaced by a TraitImpl node. This node connects a specific type to a specific trait and serves as a container for the implementations of all associated items (functions, types, constants).
2. **Model Trait Definitions:** Trait nodes should be created, containing their full definition, including associated items and supertrait bounds (e.g., trait Ord: Eq).
3. **Resolve Method Calls Dynamically:** When a query needs to resolve a method call like foo.bar(), it must replicate rustc's resolution logic. This involves first checking for inherent methods on the type of foo, and then iterating through all traits in the current scope to find a trait with a bar method for which an implementation exists for foo's type. This process is detailed within rust-analyzer's method_resolution.rs.[2]

This unlocks the ability to reason about polymorphism and answer complex questions about code behavior:

- "For a generic function parameter t: T where T: Into<String>, what are all the possible concrete types of t passed from call sites across the entire codebase?"
- "This method call resolves to a trait method. Show me the specific impl block that

provides this implementation."
- "Is this trait object-safe?"

2.3 Modeling Unsafe Rust Contexts

A complete semantic model of Rust must acknowledge the existence of unsafe Rust, which operates as a distinct superset of the language with its own rules and guarantees.[5] rust-analyzer's HIR explicitly models the safety of functions and traits via a Safety enum (Safe or Unsafe) and provides diagnostics for misuse of unsafe operations.[2]

To enable security auditing and FFI analysis, the ISG must model unsafe contexts explicitly:

1. **Tag Functions and Traits:** Add a safety: Safety field to NodeData for functions and traits.
2. **Identify Unsafe Blocks:** Introduce a new UnsafeBlock node type that can be associated with a function's body, delineating the exact regions where unsafe operations are permitted.
3. **Categorize Unsafe Operations:** Introduce new edge types or node properties to represent the five "unsafe superpowers": CallsUnsafeFn, DerefsRawPtr, AccessesMutStatic, ImplementsUnsafeTrait, and AccessesUnionField.

This provides the necessary foundation for critical security and correctness analyses:

- "Find all safe functions that internally call unsafe functions." This identifies abstraction boundaries that require careful auditing.
- "Trace all raw pointer dereferences back to their origin."
- "Generate a graph of all code reachable from FFI (Foreign Function Interface) entry points."

ISG Enrichment Layer	Proposed ISG Feature	Analogous rust-analyzer Concept	Reasoning Unlocked
Structural Semantics	Structured Visibility enum	hir::Visibility [3]	Precise API surface analysis, encapsulation checks.
	Structured	hir::Attrs,	Querying for

	Attribute representation	hir::HasAttrs [3]	metadata (#[deprecated], #[test]), doc extraction.
	CfgExpr graphs and query contexts	cfg crate [2]	Analysis of code under different features and target configurations.
Type & Trait Semantics	Per-expression inferred hir::Ty	hir-ty::infer module [2]	Concrete type analysis, generic instantiation tracking.
	TraitImpl nodes and method resolution	hir-ty::traits, next_solver [2]	Correct resolution of polymorphic code, trait usage analysis.
	Safety enum and UnsafeBlock nodes	hir::Safety, missing_unsafe diagnostic [2]	Security auditing, FFI boundary analysis, unsafe code verification.
Flow & Operational Semantics	Per-function Control Flow Graphs (CFGs)	MIR BasicBlock and Terminator [7]	Path-sensitive analysis, dead code detection, panic path analysis.
	Place-based data-flow tracking	MIR Place and Rvalue [8]	Precise taint analysis, alias analysis, impact analysis.

Section 3: Granular Flow Analysis via a Mid-level IR (MIR) Analogue

The user's proposed "fingerprints" and "stubs" are high-level summaries of a function's behavior. To achieve the deepest level of reasoning, the ISG should not just store these summaries but also the low-level representation from which they are derived. This involves lowering function bodies into a Mid-level Intermediate Representation (MIR) analogue, inspired by the architecture of rustc and rust-analyzer.[2] MIR makes control flow, data flow, and unwinding behavior explicit, transforming the ISG from a descriptive tool to a profoundly analytical one.

3.1 Explicit Control Flow Graphs (CFGs)

Analyzing the AST for control flow is notoriously difficult due to Rust's rich syntactic sugar (e.g., for loops, ? operator, match expressions). MIR solves this by desugaring all control flow into a simple, uniform Control Flow Graph (CFG).[7]

For each Function node in the ISG, a full CFG should be attached. This CFG would consist of:

- **BasicBlock nodes:** Each containing a sequence of statements.
- **A single Terminator:** Located at the end of each BasicBlock, this instruction dictates where control flow goes next.

The TerminatorKind enum from rustc's MIR provides a complete blueprint for the types of control flow transitions that must be modeled as edges in the CFG.[12]

TerminatorKind Variant	Description in Control Flow	Generated CFG Edges (Success)	Generated CFG Edges (Unwind/Panic)
Goto	An unconditional jump to another basic block.	Unconditional edge to the target block.	None.
SwitchInt	A multi-way branch based on an integer value (used for match).	One edge for each value, plus an "otherwise" edge.	None.
Call	A function call.	Edge to the return	Edge to the unwind

		block where execution continues after the call.	block, taken if the called function panics.
Assert	An assertion that a condition is true.	Edge to the success block if the condition holds.	Edge to the unwind block if the assertion fails.
Return	Returns control to the caller.	Terminal edge (no outgoing success edge within the function's CFG).	None.
Unreachable	Indicates a point in the code that should never be reached.	Terminal edge.	None.
Drop	Drops a local variable, potentially running custom drop logic.	Edge to the target block after the drop completes.	Edge to the unwind block if the drop implementation panics.

By modeling the full CFG, the ISG enables true path-sensitive analysis, allowing it to answer questions like: "Is this line of code reachable?" (dead code analysis) or "What conditions must be true for this panic! to be reached?".

3.2 Precise Data-Flow and Memory Modeling

The user's concept of "Data-Flow Stubs" can be radically improved by adopting MIR's model of memory locations, known as Places.[8] A Place is not just a variable; it is a precise path to a memory location, composed of a base local variable and a projection of field accesses, dereferences, or array indices. Operations on data are represented by Rvalues, which read from Operands (which can be Places).[9]

This allows for the replacement of high-level stubs with explicit, fine-grained DataFlow edges between MIR-level statements. The source and destination of these edges would be Places.

For example, the statement y.bar = x.foo; would generate a data-flow edge from the Place representing x.foo to the Place representing y.bar.

PlaceElem Variant	Syntax Example	Memory Location Semantics
Field	x.foo	Accesses the memory of field foo within the struct x.
Deref	*p	Accesses the memory location pointed to by the raw pointer or reference p.
Index	a[i]	Accesses the memory of the i-th element of the array or slice a.
ConstantIndex	a	Accesses the memory of the 3rd element of the array a.
Subslice	&a[1..4]	Refers to a slice of memory within the array or slice a.
Downcast	if let E::V(x)	Narrows a place of an enum type to a specific variant, enabling access to its fields.

This level of precision is the foundation for sophisticated data-flow analyses like:

- **Taint Analysis:** "If this function parameter is tainted, trace all Places that its data flows into, including through pointer dereferences and field assignments."
- **Alias Analysis:** "Which Places in this function might refer to the same memory location as *x.ptr?"

3.3 Modeling Panic and Unwind Paths

The user's Panics fingerprint can be formalized and integrated directly into the CFG. As shown in the TerminatorKind table, MIR makes unwind paths explicit. Any operation that can panic—a function call, an assertion, division, or an explicit panic!—is represented by a terminator with two potential outgoing paths: a success path and an unwind path.[12]

By modeling these unwind edges, the ISG captures the program's exception handling behavior. This is critical for reliability and correctness analysis, enabling queries such as:

- "Does this function guarantee that a resource (e.g., a file handle) is always closed, even in the event of a panic from a called function?"
- "Identify all functions that can panic, either directly or by calling another function that can panic."

Section 4: Synthesis and Advanced Reasoning Capabilities

The integration of these three layers of semantic information—structural (HIR-def), type-theoretic (HIR-ty), and operational (MIR)—transforms the ISG into a hyper-enriched semantic model capable of answering complex, cross-cutting queries that were previously intractable.

4.1 A Unified Query Model for Multi-Layered Reasoning

The true power of this enriched model lies in the ability to compose queries across its different layers. An analyst is no longer limited to asking about syntax or types in isolation but can now explore their intricate interplay.

- **Example Query 1 (Security Analysis):**
  - **Question:** "Find all public-facing functions that accept raw byte slices, where that data can flow into a raw pointer dereference within an unsafe block, when compiled for a Linux target."
  - **ISG Traversal:**
    1. Filter for Function nodes with Visibility::Public.
    2. Filter those functions for a CfgExpr compatible with a Linux target.
    3. Analyze their signatures for parameters with the type &[u8].

4. For each matching function, traverse its MIR-level data-flow graph to see if data from the parameter's Place can reach a Place that is the operand of a DerefsRawPtr operation.
5. Confirm that the DerefsRawPtr operation occurs within an UnsafeBlock node.

- **Example Query 2 (Architectural Refactoring):**
  - **Question:** "We are considering changing the associated type <T as MyTrait>::Item. Show all concrete types used for Item throughout the codebase."
  - **ISG Traversal:**
    1. Find the Trait node for MyTrait.
    2. Find all TraitImpl nodes that implement MyTrait.
    3. For each TraitImpl, extract the concrete type provided for the Item associated type.
    4. Additionally, find all Calls edges to generic functions with a T: MyTrait bound. For each call, use the stored GenericArgs to find the concrete type C substituted for T, then resolve <C as MyTrait>::Item.

4.2 Strategic Recommendations for Implementation

Implementing this entire roadmap is a significant undertaking. A phased, strategic approach is recommended to deliver value incrementally while managing complexity.

1. **Phase 1: Implement the HIR-def and HIR-ty Layers.** Focus on Sections 1 and 2. Building out structured representations for visibility, attributes, cfg, types, and traits provides immense immediate value for high-level semantic and architectural queries. This phase elevates the ISG from a syntactic to a semantic graph.
2. **Phase 2: Implement the MIR Layer.** After the HIR is stable, focus on Section 3. Develop the lowering pass from the AST/HIR to the MIR-like representation. The initial goal should be to build the CFG with correct Terminators. Place-based data-flow analysis can be added subsequently.

Throughout this process, performance must be a primary consideration. The parseltongue project's strict performance contracts preclude computing all this information eagerly. The architecture of rust-analyzer, which relies on the salsa incremental computation framework, provides a crucial lesson: **compute lazily**. The full MIR-level analysis for a function is expensive and should only be performed on-demand when a query explicitly requires it. The results can then be cached for subsequent queries, balancing responsiveness with analytical depth. This on-demand computation strategy is key to building a tool that is both powerful and interactive.

Works cited

1. rust-lang/rust-analyzer: A Rust compiler front-end for IDEs - GitHub, accessed October 13, 2025, https://github.com/rust-lang/rust-analyzer
2. rust-lang-rust-analyzer-8a5edab282632443.txt
3. hir - Rust, accessed October 13, 2025, https://rust-lang.github.io/rust-analyzer/hir/index.html
4. Visibility in syn - Rust - Docs.rs, accessed October 13, 2025, https://docs.rs/syn/latest/syn/enum.Visibility.html
5. Unsafe Rust - The Rust Programming Language - Rust Documentation, accessed October 13, 2025, https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html
6. Unsafe in Rust: Syntactic Patterns, accessed October 13, 2025, https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax/
7. The MIR (Mid-level IR) - Rust Compiler Development Guide, accessed October 13, 2025, https://rustc-dev-guide.rust-lang.org/mir/index.html
8. Place in rustc_middle::mir - Rust, accessed October 13, 2025, https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/struct.Place.html
9. Rvalue in rustc_middle::mir - Rust, accessed October 13, 2025, https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/enum.Rvalue.html
10. Charon: An Analysis Framework for Rust - arXiv, accessed October 13, 2025, https://arxiv.org/html/2410.18042v2
11. 1211-mir - The Rust RFC Book, accessed October 13, 2025, https://rust-lang.github.io/rfcs/1211-mir.html
12. TerminatorKind in rustc_middle::mir - Rust - Rust Documentation, accessed October 13, 2025, https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/enum.TerminatorKind.html