TFL 6

Evaluating the implementation of lints for Rust based on graph database queries.

Fridtjof Peer Stoldt

April 28, 2022

Contents

List of Figures

Acronyms

AST abstract sytax tree. 3–5, 7

CFG control-flow graph. 4, 5, 7

HIR high-level intermediate representation. 5, 7–11

IR intermediate representation. 1, 3–7, 11

MIR mid-level intermediate representation. 5

THIR typed high-level intermediate representation. 5

1 Introduction

Rust is a programming language that focuses on stability, reliability, and performance. The official compiler *rustc*[1] uses compile-time checks to ensure reliability and memory-safety. [20] The language is designed to enable extensive static code analysis to catch mistakes during compilation. The official linter *Clippy* provides additional code analysis and assistance to users [18]. The language and other official projects are open source and generally dual-licensed under the MIT and Apache 2.0 license [19].

1.1 Problem

In the Rust community, several members have expressed interested in developing lints that are targeted towards individual projects or frameworks. Clippy avoids these types of lints as they are usually higher in maintenance and target a smaller group of users, than general language related analysis. If users would like to develop very specific lints, they are therefore forced to write their own linter. This can be done by using the linting interface of rustc. However, this interface is highly unstable and requires constant maintenance. An additional obstacle is the interface complexity. The compiler design focuses on speed and code translation and not simplicity. Currently, there is no simple and stable way to implement lints for the Rust programming language. [9]

1.2 Research Question

The described problem in section 1.1 leads to the question: *How to design a simple and stable linting interface for Rust?* This question has been discussed in the community and was taken up by a working group outside the Rust organization. During the discussion, a user named *HeroicKatora* suggested implementing a query-based interface. [9] This paper investigates the usage of graph databases for linting in Rust to evaluate a query like interface with the research question: *Is it practical to implement lints for Rust using graph database queries?*

1.3 Aim

The primary goal of this paper is to evaluate the usage of graph database queries for linting in Rust. The process of linting for this paper involves the export of a graph based intermediate representation representing the source code. Identifying patterns using the exported data and then creating diagnostic instances that can be passed back to rustc. Including the graph export and diagnostic emission in the analysis ensures that the evaluated process can be used in practice.

1.4 Methodology

The start of this paper introduces the theoretical concept of graph databases and graph based intermediate representations. Section 4 analyses the interface provided by rustc and constructs an implementation plan to develop a prototype for further evaluation. The prototype will be used to reimplement a lint from Clippy with a graph query. The new implementation will be analyzed to evaluate the simplicity and practicality of using graph databases for linting Rust source code. The paper will concludes with a summary and further research potential related to the research question.

[1] The official name starts with a lower-case letter, see `https://doc.rust-lang.org/rustc/what-is-rustc.html` (Accessed: 2021-11-17)

2 Graph Database

A *graph database* (short for *graph database management system*) provides Create, Read, Update, and Delete (CRUD) operations for a graph data model. Graphs, based on the graph theory in mathematics, consist out of *nodes* and *edges*, also often referred to as *relationships*. Both nodes and edges are *first-class citizens* in a graph data model, meaning that they are entities with standard operations that can be performed on them. A graph database actually stores graphs as connected data. This contrasts with other database concepts, where data is implicitly connected and require additional processing to be retrieved. [10, p. 1, 5, 6, 19]

The specific graph model used by these systems can vary, the most common form is the *labeled property graph model*. A labeled property graph consists out of nodes and relationships. Both of them can have properties which are stored as key-value pairs. Nodes can additionally have labels, used for categorization and identification. All relationships are directed, named and have a start and end node. [10, p. 4] An example labeled property graph is visualized in figure 1.

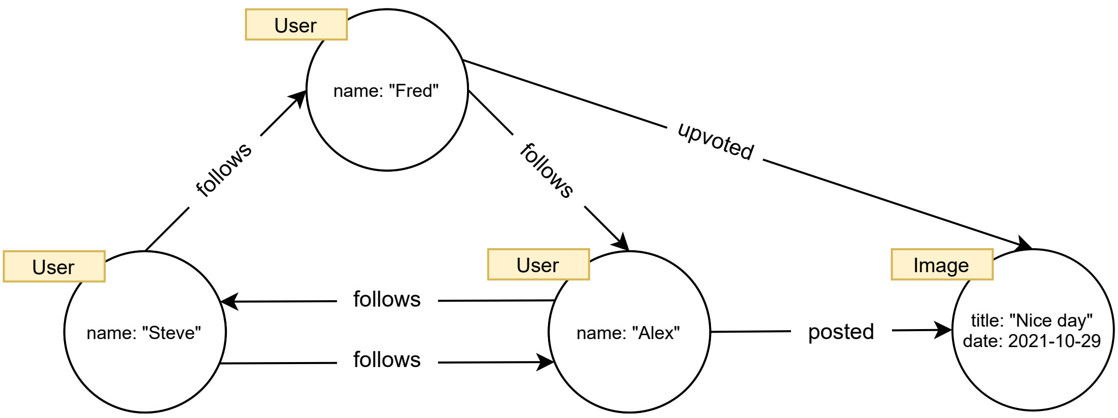

Figure 1: Labeled property graph example

Over that past years, several graph database engines for labeled property graphs have been developed. With these multiple query languages have been introduced which vary based on the implementation, expressiveness, purpose and style. The three most common languages *RDF Query Language*, *Cypher* and *Gremlin* all support the same basic operations of *graph pattern matching* and *graph navigation*. [2, p. 8]

Basic Graph pattern matching allows the selection of a node, based on the surrounding graph structure. The query is based on a single node and optionally includes label requirements for selected relationships and nodes. Graph pattern matching can be expanded with additional functionality for projections, unification, optional patterns and determining differences. These functions allow the refinement of searches and increase the precision of results. Basic graph pattern matching with additional features are referred to as *complex graph patterns*. [2, p. 8, 9]

The scope of *graph pattern matching* is bound to the described graph structure around a single node. *Graph navigation* allows defining graph patterns for connected nodes and further navigation. The navigated path between nodes can have an arbitrary length. [2, p. 21, 22]

3 Intermediate Representation (IR)

Compilers and programs that work with source code usually do not operate directly on the source code itself. They instead use intermediate representations (IR) with different levels of abstraction. [3, p. 1] Another benefit of using IRs is that these are usually independent of code formatting, which allows the analysis and transformation only based on the syntax and semantics. Doing these on the actual code is error prune [12]. This section summarizes the concept of two graph-based IR. The construction process of these representations is not documented as part of this paper.

3.1 Abstract Syntax Tree (AST)

An abstract sytax tree (AST) is an abstract representation of the syntactic structure of source code in the form of a tree graph. Each node represents an item based on the formal definition of the language and the use case that the AST is intended for. Edges between these nodes represent the hierarchical structure and logical connection of items. [16, p. 22, 26] Figure 2 is an example AST visualization of the expression `3 * 3 + (5 - 2)`. The root of the tree is the + expression, as this is also the last one that will be evaluated. The * and - nodes are both sub nodes which intern have the integer literals as leaves.

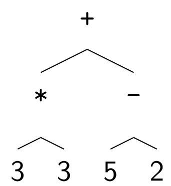

Figure 2: An example AST for the expression: `3 * 3 + (5 - 2)`

Branching expressions like loops and if statements have all branches as children, even if they are unreachable [16, p. 28]. The existence inside the graph therefore does not imply that each expression will be executed. Figure 3 illustrates an AST for an `if` expression with two branches. However, most languages will only evaluate one branch based on the condition evaluation.

```
if true then
    return 0
else
    return 5
end
```

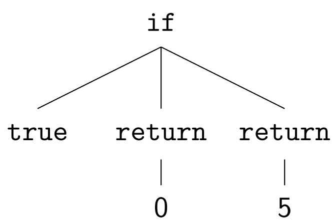

Figure 3: An example `if` expression with a corresponding AST

3.2 Control-Flow Graph (CFG)

The execution order of a program can be represented as a directed and connected graph called the control-flow graph (CFG). Compilers use these graphs for data flow analysis and performance optimization. The nodes of a CFG represent basic building blocks of a program, like single statements. Edges connecting the nodes represent the possible execution paths that a program can take. [1, p. 2] Figure 4 shows an example program and the corresponding CFG.

```
if false then
    print "hello"
end

print " "

for x in 1..3 do
    print "world"
end

print "!"
```

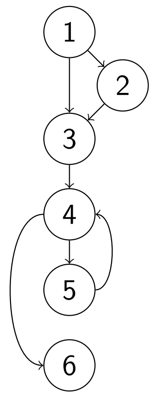

Figure 4: An example program with a corresponding CFG

4 Linting with rustc

The aim of this paper is to evaluate the usability of graph databases for linting Rust code. This section introduces the linting interface provided by rustc and the intermediate representation used inside the compiler. The concepts behind the IRs are documented in section 3. The insights from this chapter are then used to create a prototype for linting with graph database queries.

4.1 Intermediate Representations in rustc

The official Rust compiler uses several stages to translate source code to a defined IR, which can then be passed to different backends. This structure allows the compiler to target several platforms like LLVM and web assembly. Stages use different IRs that are specific for their intended purpose.[2] The relevant IRs as of 2021-11-17 are:

- **abstract sytax tree (AST)**

  The compilation in rustc starts with the translation of source code into an AST this IR is also called *AST* inside the compiler. Rust supports macros which are expanded as part of this process. [12] The AST can be assessed before and after the expansion of these macros [21].

[2] The documentation for elements used to construct these intermediate representations can be found under `https://doc.rust-lang.org/nightly/nightly-rustc/` (Accessed: 2021-11-17).

- **high-level intermediate representation (HIR)**

  The high-level intermediate representation (HIR) is the central IR used by rustc. The HIR is still an AST representation, where all identifiers have been resolved, macros have been expanded, and some expressions have been desugared. The compiler can resolve expression types during this stage of compilation. Each node in this AST has a unique identifier, called `HirId`, which can be accessed and referenced in several ways. [13]

- **typed high-level intermediate representation (THIR)**

  The typed high-level intermediate representation (THIR) is created once type checking is completed. THIR is derived from HIR but only contains executable code, without additional information like item definitions. This IR is mainly used to create the next IR from the HIR tree. For this translation, the type of every expression is evaluated. The elements of this IR are separated into different arrays and dropped as soon as they are no longer needed. THIR does not directly represent the program as a graph. [14]

- **mid-level intermediate representation (MIR)**

  The MIR is the last IR used inside rustc before a IR resembling machine code is passed to the selected backend. MIR is a very simplified version of the program in the form of a control-flow graph described in section 3.2. This form allows flow-sensitive code analysis and optimizations, which are complicated to perform on an AST. This IR is used inside the compiler for code analysis. [7]

From these IRs the following once have a graph like structure and can therefore be considered in this paper: AST, HIR and MIR.

4.2 rustc's Linting Interface

Rustc provides a linting interface for internal and external use. The interface provides access to the AST before and after the expansion of macros and the HIR tree. [11] The functionality to lint the AST before the expansion of macros is softly deprecated and should therefore be avoided [6] [4]. The structure and purpose of these IRs is documented in section 4.1. The compiler internally traverses the respective graphs and passes each node in order to registered callbacks, called lint passes. Linting code that implements this callback can then check the respective nodes and access additional information to check for violations. [11]

Lint and error messages in rustc are emitted as diagnostic instances. These allow internal and external code to leverage rustc's infrastructure for message emission[3]. A diagnostic instance consists out of a level which indicates the severity, optionally an error code, a message describing the problem and a span marking the suspicious code. The source code marked by the diagnostic span is displayed as part of the user output, along with the file location. Messages can additionally contain, explanatory comments and suggestions how the problem can be corrected[4]. [11] The console output of an emitted diagnostic can be seen in figure 5.

[3]Internal diagnostics in rustc can be emitted without using the official linting interface. Using this interface reduces the complexity and is therefore advisable.

[4]The structure of the `Diagnostic` struct is documented under: `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_errors/diagnostic/struct.Diagnostic.html` (Accessed: 2021-11-20)

```
error: calls to ‘push‘ immediately after creation
  --> $DIR/query_vec_init_then_push.rs:5:5
   |
LL | /     let mut def_err: Vec<u32> = Default::default();
LL | |     def_err.push(0);
   | |__________________^ help: consider using the ‘vec![]‘ macro: ‘let mut def_err = vec![..];‘
```

Figure 5: Example of rustc's console output of lint diagnostics

Suggestions for diagnostic messages are usually created by copying the original source code using spans and then creating a suggestion out of them. The span for the creation is extracted from the nodes of each IR. All suggestion includes a confidence called *applicability*. The highest confidence allows the automatic application of the suggested change. [11]

The linting process described in section 4.2 is roughly structured into three phases. First, the linting logic receives nodes of the IR which are then analyzed for suspicious patterns. This can include checking connected nodes, node properties and expression types. Secondly, the creation of a code suggestion. This step is optional and only done if a sensible sugge

To evaluate the practicality and simplicity of graph database queries for linting, a prototype will be developed. This section describes the selected design, implementation details and conclude with an analysis. For the evaluation, a lint from Clippy will be reimplemented with the prototype.

4.3 Design

This subsection documents the design of the developed prototype. The design will not target a specific lint implementation, but rather the general concept of integrating a graph database into the linting process.

4.3.1 Graph Database

The prototype developed as part of this paper will use *Neo4j* as a graph database. Neo4j uses labeled property graphs as described in section 2. *Cypher* is used as the native graph query language for Neo4j and is used inside the prototype. The language supports complex graph pattern matching and graph navigation, as described in section 2. The concept evaluated in this paper using Cypher should be transferable to other graph query languages supporting these features.

4.3.2 Stage of the Linting Process

The linting process described in section 4.2 is roughly structured into three phases. First, the linting logic receives nodes of the IR which are then analyzed for suspicious patterns. This can include checking connected nodes, node properties and expression types. Secondly, the creation of a code suggestion. This step is optional and only done if a sensible suggestion can be created. Lastly, if a lint violation was detected, the creation and emission of a diagnostic instance. This step requires the source node where the diagnostic should be emitted, the affected span and a lint message. Additional notes and help messages can be attached during this step. [17]

The prototype developed for this paper tries to implement the first phase of the node analysis with a graph database. For this, the IR will be exported to the database as a labeled property graph. After the completed export, a graph query is used to detect suspicious graph structures which would usually be evaluated in Rust. The query then returns all relevant information to continue the linting process. One limitation of this model is that the graph database query will only have access to the exported data. Helper functionalities provided by rustc and Clippy are inaccessible. This design might therefore also require additional checks once a suspicious graph pattern is found. The following two steps of creating a suggestion and emitting the diagnostic instance have to be implemented in Rust, as they usually require interfacing with rustc directly.

4.3.3 Target Intermediate Representation

Section 3 introduces two graph-based IRs which are both used inside rustc. Both represent different aspects of a program. A CFG enables control flow analysis and performance optimizations. Related lints are implemented in rustc itself. For lints related to language usage, syntactic and semantic analysis, the AST representation is better suited. The prototype uses the official linting interface of rustc, as it provides access to all AST-based IRs.

Rust uses a strong and static type system to validate the correctness of a program at compile time [5, p. 9]. In Rust, type information express a part of the program semantics [15]. The type information of expressions is available after the HIR tree has been constructed. For this reason, the HIR will be used for the remainder of this paper. It contains all information of the AST and provides additional node information. It is also the main IR used inside Clippy. An additional benefit is the unique identifier defined in this stage. The `HirId` can be returned by the graph query to identify and retrieve the node from rustc again.

The HIR tree will need to be transformed into a labeled property graph to export it to the graph database. The HIR nodes have properties and child nodes, which are identified by their order and type. The prototype implementation exports the properties as node properties. Child nodes are attached with a child relationship which holds an index property.

Additional node information like the expression type is not exported, as the type representation is too complex for the scope of this paper. Type and possibly some additional checks will be done in Rust after a suspicious graph pattern has been identified. This chosen solution will be reflected on as part of the prototype evaluation in section 4.6.

The graph will be constructed will be constructed with a lint pass. Each node is then visited individually to create a corresponding node and relationship in the graph database. The linting query will be executed after each function was exported, as rustc's type context is function-specific. It can only be accessed while the lint pass is technically inside a function body.

4.4 Framework Implementation

The prototype build as part of this paper uses the HIR as an intermediate representation with the structure of an AST. The linting prototype will be registered as a lint pass in the linting interface of rustc. Nodes and relationships provided by rustc will be translated into a labeled property graph model for the graph database Neo4j using Cypher as a query language.

Expression types are not translated into a graph or property representation. Type checks are done after a pattern in the graph has been identified. Relationships between nodes are identified with an index during the export.

The linting process of the prototype is done in stages. First, the HIR tree is translated into a labeled property graph. Afterwards, a Cypher graph query is used to identify suspicious graph patterns. The identified nodes are returned to the lint pass for type checking if required. After the successful identification, a diagnostic object will be created and emitted to rustc's linting interface.

This design and structure is based on section 4.3. The graph creation source code is provided in appendix A. The evaluation is done by reimplementing a Clippy lint using a graph query. The new lint implementation has to pass all tests inside the project. The two implementations will be compared afterwards.

The design could be implemented as planned. The only problem arose during the exportation of the `HirIds`. The identifier internally uses two unsigned 32-bit integers, that are not publicly accessible. The prototype transmutes the `HirId` into integers to export them. This is an unsafe operation that violates the interface of rustc. The transmutation has to be reversed for the creation of `HirIds` as no constructor is accessible outside the compiler. The violation of the interface and usage of an unsafe operation is an accepted risk as part of this prototype.

4.5 Lint Implementation

For the evaluation, a lint from Clippy will be reimplemented with a graph query language. Clippy provides over 450 lints as of 2021-11-11 [18]. The `clippy::vec_init_then_push` lint was selected for the evaluation as it uses graph pattern matching and requires only one type check. The referenced implementation inside Clippy was developed by Jason Newcomb. The lint detects new `std::vec::Vec<_>` instances that are immediately used to push new values into. The emitted diagnostic suggest using the `vec![]` macro, as it is faster and easier to read. The Rust implementation first finds variables that are initialized by a new `std::vec::Vec<_>` instance. The logic then continues to check if the following statements push values into the newly created instance. [8]

The reimplementation has to find value assignments that are used directly afterwards for a push operation. The verification that the value is actually a `std::vec::Vec<_>` instance is done afterwards. The query returns the `HirId` of the assignment and method call expressions. The Rust code additionally has to ensure that both statements use the same variables.

A value assignment in Rust can have two HIR tree representations, depending on whether the variable is newly created or reassigned. Here, both cases are covered by one graph pattern, as both representations only differ in the node labels and properties. Combining these into one cypher query increases the complexity of the *where* condition in the query. The query to match all value assignments is displayed in figure 6. Retrieving the next node and verifying that it is a method call to a `push` function can be archived with graph navigation. The complete Cypher query to find suspicious graph patterns can be found in appendix B.

```
MATCH
    (assign {from_expansion: false}),
    (assign)-[:Child {index: 0}]->(var),
    (assign)-[:Child {index: 1}]->(init_call:Call)-[:Child]->(init:Path)
WHERE
    (var.name = "Pat" OR var.name = "Path")
    AND (assign.name = "Local" OR assign.name = "Assign")
    AND (init.path CONTAINS 'new'
        OR init.path CONTAINS 'with_capacity'
        OR init.path CONTAINS 'default')
return var, assign, init_call, init
```

Figure 6: The Cypher query used to match variable assign graph patterns

The original lint spans over all following push operations. Implementing this in Cypher would require returning several identified push operations. This functionality is limited by the fact that all return values have to be declared inside the query. Having a fixed amount of returned values limits the following Rust code and might require additional logic.

Detected code patterns are further processed in Rust. The returned `HirIds` are used to retrieve the actual HIR nodes from rustc. These are needed for type checking and to determine the expression span. The expressions are also required to verify that they both reference the same variable. This check can potentially be done in the graph if path expressions get resolved during the graph creation.

The diagnostic emission is almost identical with the original Rust implementation. The difference only arises from the lint name and reduced span, as only one following push statement is marked in the diagnostic. The complete lint code is found in appendix C.

4.6 Evaluation

The `clippy::vec_init_then_push` lint was successfully implemented with the usage of a graph database query. The new implementation correctly detects all instance in the test file of the original lint and emits a related message. The created diagnostic displays the assign statement and the first `push` call. The rest of this section will evaluate the implementation in comparison to the Clippy version[5].

One advantage of a graph query implementation is the provided tooling for graph databases. With these tools, it is possible to visualize the entire HIR tree and directly check query results. This stance in contrast to rustc which only provides a textual representation of the tree. During lint development, it is required to compile and run Clippy to test if all patterns are detected correctly. Debugging a graph query itself is therefore faster using the provided tools than using a pure Rust implementation.

In rustc, some semantically equivalent expressions can have different representations inside the HIR graph. The two important cases for the reimplemented lint could be combined into one query. However, this will not be possible for every expression. The difference between control flow statements like `if` and `match` cannot simply be combined into one statement due to their different graph structure. Clippy provides

[5]The Clippy lint implementation can be found under `https://github.com/rust-lang/rust-clippy/blob/cc9d7fff/clippy_lints/src/vec_init_then_push.rs` (Accessed: 2021-11-18)

wrapper objects to handle both these cases alike if needed. These wrappers are not usable in graph query languages.

The exported labeled property graph only contains linting related information. The query only returns the node `HirId` to allow further processing. The internal representation of the `HirId` is not publicly accessible. For this prototype, an unsafe transmute function is used. If graph queries should be used in practice, an interface has to be added to avoid this violation of the interface. The transmutation usage caused no problem during the evaluation.

The benefit of using a graph query for the graph pattern matching is limited by the fact, that most nodes needed to be retrieved from rustc afterwards. The nodes where required for span data, type checking and path resolution. The required overhead in this prototype is about the same length as the entire pattern matching in Clippy's implementation. More information could be exported as part of the labeled property graph to reduce additional checks in Rust. The practicality of a graph database is still reduced by this.

The prototype used Neo4j as a graph database. The use of an external system like Neo4j creates a communication overhead, as two systems have to exchange information. The performance was not measured as part of this prototype. However, the minimum communication error handling used for this paper is with 15 lines, a noticeable part of the implementation.

4.7 Analysis Result

The lint implementation 4.5 proves that a graph databases can be used as part of the linting process with rustc. The evaluation in section 4.6 found several downsides related to the usage of graph queries for linting. Only one real advantage was found related to the usage. Based on this evaluation, the research question defined in 1.2 concludes that the usage of a graph database for linting Rust code is neither simple nor practical.

5 Conclusion

This paper succeeded to evaluate the usage of a graph database for linting Rust code. A lint was successfully recreated to assess the practicality of a graph query language for linting. It was determined that the usage of a graph database is impractical with the current linting interface of rustc.

5.1 Summary

Section 1 introduced the Rust project and the general interest of the community to write custom lints. The linting interface of the official compiler is highly unstable. The community has started to discuss alternative options to provide an interface for linting. This paper investigates the research question: *Is it practical to implement lints for Rust using graph database quieries?*

A prototype using rustc's linting interface was developed to assess the usage of graph database queries for linting Rust code. The HIR tree of rustc is exported as a labeled property graph to a Neo4j database. The query language Cypher is used for complex graph pattern matching and graph navigation.

The prototype and lint implementation are analyzed in section 4.6. The assessment concluded that the usage of graph databases is impractical. The only found advantage is the visualization and debugging support provided by graph database tool. In contrast, several downsides were found. The HIR graph export and import use unsafe code and violate the interface of rustc to access the unique node identifier. Identified nodes needed to be retrieved from the compiler again for type checking and the creation of a diagnostic object. This limits the usefulness of finding suspicious code pattern with Cypher beforehand. The usage of Neo4j as an external system additionally added a communication overhead. The analysis concluded that the disadvantages outweigh the found benefit.

5.2 Outlook

This paper used an external system for graph pattern matching and graph navigation. Most found issues arise from the usage of this external system. A Rust implementation for these functionalities can potentially avoid these disadvantages. It is still questionable if a different interface to rustc's IRs would improve the practicality of using a graph-query-like approach.

Another option could be to create a cross-compiler to translate graph queries into Rust code. Clippy already supports some automatic lint logic creation with the `clippy::author` attribute [17]. This concept can potentially be expanded to support graph navigation.

References

[1] Frances E. Allen. “Control Flow Analysis”. In: *SIGPLAN Notices* 5.7 (1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: https://doi.org/10.1145/390013.808479.

[2] Renzo Angles et al. “Foundations of Modern Query Languages for Graph Databases”. In: *ACM Computing Surveys* 50.5 (2017). ISSN: 0360-0300. DOI: 10.1145/3104031. URL: https://doi.org/10.1145/3104031.

[3] Fred Chow. “Intermediate Representation: The Increasing Significance of Intermediate Representations in Compilers”. In: *Queue* 11.10 (2013), pp. 30–37. ISSN: 1542-7730. DOI: 10.1145/2542661.2544374. URL: https://doi.org/10.1145/2542661.2544374.

[4] Mazdak Farrokhzad and The Rust Project Developers. *Expansion-driven outline module parsing*. https://github.com/rust-lang/rust/pull/69838. Accessed: 2021-11-04. 2020.

[5] Ralf Jung et al. “Safe Systems Programming in Rust”. In: *Commun. ACM* 64.4 (2021), pp. 144–152. ISSN: 0001-0782. DOI: 10.1145/3418295. URL: https://doi.org/10.1145/3418295.

[6] Philipp Krones and The Rust Clippy Developers. *[WIP] Register redundant_field_names and non_expressive_names as early passes*. https://github.com/rust-lang/rust-clippy/pull/5518. Accessed: 2021-11-04. 2020.

[7] Niko Matsakis. *Introducing MIR*. https://blog.rust-lang.org/2016/04/19/MIR.html. Accessed: 2021-11-14. 2016.

[8] Jason Newcomb and The Rust Clippy Developers. *New lint: vec_init_then_push*. https://github.com/rust-lang/rust-clippy/pull/6538. Accessed: 2021-11-18. 2021.

[9] Jacob Pratt and et. al. *Towards a Stable Compiler API and Custom Lints*. https://internals.rust-lang.org/t/towards-a-stable-compiler-api-and-custom-lints/15048. Accessed: 2021-11-17. 2021.

[10] Ian Robinson, Jim Webber, and Emil Eifrém. *Graph Databases*. 2nd Edition. O’Reilly Media, 2015. ISBN: 978-1-491-93200-1.

[11] Rustc developers. *Errors and Lints*. https://rustc-dev-guide.rust-lang.org/diagnostics.html. Accessed: 2021-11-16. 2021.

[12] Rustc developers. *Syntax and the AST*. https://rustc-dev-guide.rust-lang.org/syntax-intro.html. Accessed: 2021-11-11. 2020.

[13] Rustc developers. *The HIR*. https://rustc-dev-guide.rust-lang.org/hir.html. Accessed: 2021-11-15. 2020.

[14] Rustc developers. *The THIR*. https://rustc-dev-guide.rust-lang.org/thir.html. Accessed: 2021-11-15. 2021.

[15] Rustc developers. *The ty module: representing types*. https://rustc-dev-guide.rust-lang.org/ty.html. Accessed: 2021-11-16. 2021.

[16] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995. ISBN: 0-201-65697-3.

[17] The Rust Clippy Developers. *Adding a new lint*. https://github.com/rust-lang/rust-clippy/blob/master/doc/adding_lints.md. Accessed: 2021-11-18. 2021.

[18] The Rust Clippy Developers. *Clippy*. https://github.com/rust-lang/rust-clippy. Accessed: 2021-11-18. 2021.

[19] The Rust Project Developers. *Licenses*. `https://www.rust-lang.org/policies/licenses`. Accessed: 2021-10-26.

[20] The Rust Project Developers. *Rust*. `https://www.rust-lang.org/`. Accessed: 2021-10-26. 2021.

[21] The Rust Project Developers. *Struct rustc_lint::LintStore*. `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_lint/struct.LintStore.html`. Accessed: 2021-11-18.

A Labeled Property Graph Creation Prototype

`````
use clippy_utils::diagnostics::span_lint_and_sugg;
use clippy_utils::source::snippet;
use clippy_utils::ty::is_type_diagnostic_item;
use clippy_utils::{is_lint_allowed, path_to_local, path_to_local_id};
use rustc_errors::Applicability;
use rustc_hir as hir;
use rustc_hir::intravisit::FnKind;
use rustc_lint::{LateContext, LateLintPass};
use rustc_session::{declare_lint_pass, declare_tool_lint};
use rustc_span::symbol::sym;
use rustc_span::Span;

use rusted_cypher::cypher::result::Row;
use rusted_cypher::{cypher_stmt, GraphClient, GraphError, Statement};

const DB_URL: &str = "http://neo4j:pw-clippy-query@localhost:7474/db/data";

// [...] Lines 17 - 37
// Not part of the actual graph creation

declare_clippy_lint! {
    /// ### What it does
    ///
    /// ### Why is this bad?
    ///
    /// ### Example
    /// ```rust
    /// // example code where Clippy issues a warning
    /// ```
    /// Use instead:
    /// ```rust
    /// // example code which does not raise Clippy warning
    /// ```
    pub GRAPH_QUERY_LINTER,
    nursery,
    "default lint description"
}

declare_lint_pass!(GraphQueryLinter => [GRAPH_QUERY_LINTER]);

impl LateLintPass<'_> for GraphQueryLinter {
    // A full blown linter would start at the `Crate` node with `check_crate` however this is
    // just a fancy prototype. Therefore, I'll start at the function level with `check_fn` that
    // reduces the complexity (a lot).

    /// This `check_fn` is the entry point to the graph generation later used for linting. Some
    /// arguments are ignored as they provide span or type data that will not be exported as part
    /// of the labeled property graph. They'll later be retrieved from the [`LateContext`].
    fn check_fn(
        &mut self,
        cx: &LateContext<'tcx>,
        _kind: FnKind<'tcx>,
        _fn_declaration: &'tcx hir::FnDecl<'tcx>,
        body: &'tcx hir::Body<'tcx>,
        _span: Span,
        hir_id: hir::HirId,
    ) {
        if is_lint_allowed(cx, GRAPH_QUERY_LINTER, hir_id) {
            return;
`````

```rust
        }

        let graph = GraphClient::connect(DB_URL).unwrap();
        graph.exec("MATCH (node) DETACH DELETE (node)").unwrap();

        // We'll pass the actual graph creation off to a visitor to use a stack like structure.
        // Similar to how [`rustc_lint::levels::LintLevelsBuilder`] does it.
        create_body_graph(cx, body);

        // [...] Lines 87
        // Not part of the actual graph creation
    }
}

// [...] Lines 91 - 166
// Not part of the actual graph creation

fn create_body_graph<'tcx>(cx: &LateContext<'tcx>, body: &'tcx hir::Body<'tcx>) {
    let mut query_creator = GraphCreateVisitor::new(cx);

    query_creator.visit_body(body);

    #[rustfmt::skip]
    {
        // Use
        // ```cmd
        // sudo docker run --publish=7474:7474 --publish=7687:7687 --volume=$home/neo4j/data:/data --volume=$home/neo4j/logs:/logs neo4j:3.5.29-community
        // ```
        // to create the docker container. Note that this implementation is using 3.5.
        // Keep the default neo4j user and set the password to pw-clippy-query
    }
    let graph = GraphClient::connect(DB_URL).unwrap();
    let mut query = graph.query();
    query.set_statements(query_creator.query);
    let res = query.send();
    assert!(res.is_ok(), "ERR: {:#?}", res.unwrap_err());
}

fn serialize_hir_id(hir_id: hir::HirId) -> String {
    assert_eq!(std::mem::size_of::<hir::HirId>(), 8, "The size of HirId is weird");

    // > transmute is **incredibly** unsafe. There are a vast number of ways to cause undefined
    // > behavior with this function. transmute should be the absolute last resort.
    //
    // It's gonna be fine... ~@xFrednet

    let (owner, local_id) = unsafe { std::mem::transmute::<hir::HirId, (u32, u32)>(hir_id) };

    format!("{:08X}-{:08X}", owner, local_id)
}

fn deserialize_hir_id(hir_id_str: String) -> hir::HirId {
    assert_eq!(std::mem::size_of::<hir::HirId>(), 8, "The size of HirId is weird");

    let (owner_str, local_id_str) = hir_id_str.split_once("-").unwrap();
    let owner_int = u32::from_str_radix(owner_str, 16).unwrap();
    let local_id_int = u32::from_str_radix(local_id_str, 16).unwrap();

    unsafe { std::mem::transmute::<(u32, u32), hir::HirId>((owner_int, local_id_int)) }
}
```

```

struct GraphCreateVisitor<'a, 'tcx> {
    #[allow(dead_code)]
    cx: &'a LateContext<'tcx>,
    query: Vec<Statement>,
}

impl<'a, 'tcx> GraphCreateVisitor<'a, 'tcx> {
    fn new(cx: &'a LateContext<'tcx>) -> Self {
        Self { cx, query: Vec::new() }
    }
}

#[rustfmt::skip]
impl<'a, 'tcx> GraphCreateVisitor<'a, 'tcx> {

    fn create_link(&mut self, from_hir_id: hir::HirId, edge: &str, index: usize, to_hir_id: hir::HirId) {
        let from_hir_id = serialize_hir_id(from_hir_id);
        let to_hir_id = serialize_hir_id(to_hir_id);

        self.query.push(cypher_stmt!(
            r#"
            MATCH (from), (to)
                where from.hir_id = {from_hir_id} AND to.hir_id = {to_hir_id}
            CREATE (from)-[:Child {name: {edge}, index: {index}}]->(to)"#, {
                "from_hir_id" => &from_hir_id,
                "to_hir_id" => &to_hir_id,
                "edge" => edge,
                "index" => index
            }
        )
        .unwrap());
    }

    fn visit_body(&mut self, body: &'tcx hir::Body<'tcx>) {
        // Create Self node
        let self_hir_id = body.id().hir_id;
        self.query.push(
            cypher_stmt!(
                "CREATE (:Body {name: 'body', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                    "hir_id" => &serialize_hir_id(self_hir_id),
                    "from_expansion" => body.value.span.from_expansion()
                }
            )
            .unwrap(),
        );

        // Extracting parameters
        for (index, param) in body.params.iter().enumerate() {
            self.query.push(
                cypher_stmt!(
                    "CREATE (:Param { name: 'param', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                        "hir_id" => &serialize_hir_id(param.hir_id),
                        "from_expansion" => param.span.from_expansion()
                    }
                )
                .unwrap(),
            );
            self.create_link(self_hir_id, "Param", index, param.hir_id)
        }

```

```
        let child_hir_id = self.visit_expr(&body.value);
        self.create_link(self_hir_id, "Child", 0, child_hir_id)
    }

    fn visit_expr(&mut self, ex: &'tcx hir::Expr<'tcx>) -> hir::HirId {
        let from_expansion = ex.span.from_expansion();
        let self_hir_id = ex.hir_id;
        let hir_id = serialize_hir_id(self_hir_id);

        match &ex.kind {
            hir::ExprKind::Err
                | hir::ExprKind::ConstBlock(..)
                | hir::ExprKind::Box(..)
                | hir::ExprKind::InlineAsm(..)
                | hir::ExprKind::LlvmInlineAsm(..)
                | hir::ExprKind::Struct(..)
                | hir::ExprKind::Repeat(..)
                | hir::ExprKind::Yield(..)
                | hir::ExprKind::Closure(..)
                | hir::ExprKind::Cast(..)
                | hir::ExprKind::Type(..) => unimplemented!("Ignored for the sake of simplicity {:#?}", ex),
            hir::ExprKind::DropTemps(..) => {
                // Accepted ignore
            }
            hir::ExprKind::Tup(args) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Tup {name: 'Tup', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                for (index, child) in args.iter().enumerate() {
                    let child_hir_id = self.visit_expr(child);
                    self.create_link(ex.hir_id, "Child", index, child_hir_id);
                }
            }
            hir::ExprKind::Array(args) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Array {name: 'Array', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                for (index, child) in args.iter().enumerate() {
                    let child_hir_id = self.visit_expr(child);
                    self.create_link(ex.hir_id, "Child", index, child_hir_id);
                }
            },
            hir::ExprKind::Call(call, args) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Call {name: 'Call', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
```

```
                        "from_expansion" => from_expansion
                    }
                )
                .unwrap(),
            );

            let call_hir_id = self.visit_expr(call);
            self.create_link(ex.hir_id, "Call", 0, call_hir_id);

            for (index, child) in args.iter().enumerate() {
                let child_hir_id = self.visit_expr(child);
                self.create_link(ex.hir_id, "Child", index, child_hir_id);
            }
        },
        hir::ExprKind::MethodCall(path_seg, _span1, args, _span2) => {
            self.query.push(
                cypher_stmt!(
                    "CREATE (:MethodCall {name: 'MethodCall', hir_id: {hir_id}, from_expansion: {from_expansion}, ident: {ident}})", {
                        "hir_id" => &hir_id,
                        "from_expansion" => from_expansion,
                        "ident" => &format!("{}", &path_seg.ident.as_str())
                    }
                )
                .unwrap(),
            );

            for (index, child) in args.iter().enumerate() {
                let child_hir_id = self.visit_expr(child);
                self.create_link(ex.hir_id, "Child", index, child_hir_id);
            }
        },
        hir::ExprKind::Binary(op, left, right) => {
            self.query.push(
                cypher_stmt!(
                    "CREATE (:AssignOp {name: 'AssignOp', hir_id: {hir_id}, from_expansion: {from_expansion}, op: {op}})", {
                        "hir_id" => &hir_id,
                        "from_expansion" => from_expansion,
                        "op" => &format!("{:?}", op)
                    }
                )
                .unwrap(),
            );

            let child_hir_id = self.visit_expr(left);
            self.create_link(ex.hir_id, "Child", 0, child_hir_id);

            let child_hir_id = self.visit_expr(right);
            self.create_link(ex.hir_id, "Child", 1, child_hir_id);
        },
        hir::ExprKind::Unary(op, expr) => {
            self.query.push(
                cypher_stmt!(
                    "CREATE (:AssignOp {name: 'AssignOp', hir_id: {hir_id}, from_expansion: {from_expansion}, op: {op}})", {
                        "hir_id" => &hir_id,
                        "from_expansion" => from_expansion,
                        "op" => &format!("{:?}", op)
                    }
                )
```

```
                .unwrap(),
            );

            let child_hir_id = self.visit_expr(expr);
            self.create_link(ex.hir_id, "Child", 0, child_hir_id);
        },
        hir::ExprKind::Let(pat, expr, _span) => {
            self.query.push(
                cypher_stmt!(
                    "CREATE (:Let {name: 'Let', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                        "hir_id" => &hir_id,
                        "from_expansion" => from_expansion
                    }
                )
                .unwrap(),
            );

            let pat_hir_id = self.visit_pat(pat);
            self.create_link(ex.hir_id, "Child", 0, pat_hir_id);

            let child_hir_id = self.visit_expr(expr);
            self.create_link(ex.hir_id, "Child", 1, child_hir_id);
        },
        hir::ExprKind::If(condition, then_expr, else_expr) => {
            self.query.push(
                cypher_stmt!(
                    "CREATE (:If {name: 'If', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                        "hir_id" => &hir_id,
                        "from_expansion" => from_expansion
                    }
                )
                .unwrap(),
            );

            let condition_hir_id = self.visit_expr( condition);
            self.create_link(ex.hir_id, "Condition", 0, condition_hir_id);

            let then_hir_id = self.visit_expr( then_expr);
            self.create_link(ex.hir_id, "Then", 0, then_hir_id);

            if let Some(else_expr) = else_expr {
                let else_hir_id = self.visit_expr( else_expr);
                self.create_link(ex.hir_id, "Else", 0, else_hir_id);
            }
        },
        hir::ExprKind::Loop(block, _label, _source, _span) => {
            self.query.push(
                cypher_stmt!(
                    "CREATE (:Loop {name: 'Loop', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                        "hir_id" => &hir_id,
                        "from_expansion" => from_expansion
                    }
                )
                .unwrap(),
            );

            let child_hir_id = self.visit_block( block);
            self.create_link(ex.hir_id, "Child", 0, child_hir_id);
        },
        hir::ExprKind::Match(scrutinee, arms, _source) => {
            self.query.push(
```

```
                    cypher_stmt!(
                        "CREATE (:Match {name: 'Match', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                let scrutinee_hir_id = self.visit_expr(scrutinee);
                self.create_link(self_hir_id, "Child", 0, scrutinee_hir_id);

                for (index, arm) in arms.iter().enumerate() {
                    let arm_hir_id = arm.hir_id;
                    self.query.push(
                        cypher_stmt!(
                            "CREATE (:Arm {name: 'Arm', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                                "hir_id" => &serialize_hir_id(arm.hir_id),
                                "from_expansion" => from_expansion
                            }
                        )
                        .unwrap(),
                    );
                    self.create_link(self_hir_id, "Arm", index, arm_hir_id);

                    if let Some(_guard) = &arm.guard {
                        unimplemented!();
                    }

                    let child_hir_id = self.visit_expr(arm.body);
                    self.create_link(arm.hir_id, "Child", 0, child_hir_id);
                }
            },
            hir::ExprKind::Block(block, _) => {
                return self.visit_block(block);
            },
            hir::ExprKind::Assign(value, expr, _) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Assign {name: 'Assign', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                let child_hir_id = self.visit_expr(value);
                self.create_link(ex.hir_id, "Child", 0, child_hir_id);

                let child_hir_id = self.visit_expr(expr);
                self.create_link(ex.hir_id, "Child", 1, child_hir_id);
            },
            hir::ExprKind::AssignOp(op, value, expr) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:AssignOp {name: 'AssignOp', hir_id: {hir_id}, from_expansion: {from_expansion}, op
                        ↪ : {op}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion,
                            "op" => &format!("{:?}", op)
```

```
                        }
                    )
                    .unwrap(),
                );

                let child_hir_id = self.visit_expr(value);
                self.create_link(ex.hir_id, "Child", 0, child_hir_id);

                let child_hir_id = self.visit_expr(expr);
                self.create_link(ex.hir_id, "Child", 1, child_hir_id);
            },
            hir::ExprKind::Field(value, ident) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Field {name: 'Field', hir_id: {hir_id}, from_expansion: {from_expansion}, ident: {ident}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion,
                            "ident" => &*ident.as_str()
                        }
                    )
                    .unwrap(),
                );

                let child_hir_id = self.visit_expr(value);
                self.create_link(ex.hir_id, "Child", 0, child_hir_id);
            },
            hir::ExprKind::Index(value, index) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Index {name: 'Index', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                let child_hir_id = self.visit_expr(value);
                self.create_link(ex.hir_id, "Child", 0, child_hir_id);

                let child_hir_id = self.visit_expr(index);
                self.create_link(ex.hir_id, "Index", 0, child_hir_id);
            },
            hir::ExprKind::Path(path) => {
                let path_str = match path {
                    hir::QPath::Resolved(_, path) => {
                        path.segments.iter().fold(String::new(), |mut acc, seg| {
                            acc.push_str(&seg.ident.as_str());
                            acc
                        })
                    },
                    hir::QPath::TypeRelative(ty, path_seg) => {
                        format!("{:?}::{:?}", ty, path_seg.ident.as_str())
                        // if let hir::Adt(adt, _) = ty.kind {
                        // let path = self.cx.get_def_path(adt.did).iter().fold(String::new(), |acc, sym| {
                        // acc.push_str(&sym.as_str());
                        // acc
                        // });
                        // path.push_str(&path_seg.ident.as_str());
                        // path
```

```
                        // } else {
                        // unimplemented!()
                        // }
                    },
                    hir::QPath::LangItem(..) => unimplemented!(),
                };

                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Path {name: 'Path', hir_id: {hir_id}, from_expansion: {from_expansion}, path: {path}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion,
                            "path" => &path_str
                        }
                    )
                    .unwrap(),
                );
            },
            hir::ExprKind::AddrOf(borrow_kind, mutability, value) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:AddrOf {name: 'AddrOf', hir_id: {hir_id}, from_expansion: {from_expansion}, borrow_kind: {borrow_kind}, mutability: {mutability}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion,
                            "borrow_kind" => &format!("{:?}", borrow_kind),
                            "mutability" => &format!("{:?}", mutability)
                        }
                    )
                    .unwrap(),
                );

                let child_hir_id = self.visit_expr(value);
                self.create_link(ex.hir_id, "Child", 0, child_hir_id);
            },
            hir::ExprKind::Break(dest, value) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Break {name: 'Break', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                if let Ok(target) = dest.target_id {
                    self.create_link(ex.hir_id, "Jump", 0, target);
                }

                if let Some(value) = value {
                    let child_hir_id = self.visit_expr(value);
                    self.create_link(ex.hir_id, "Child", 0, child_hir_id);
                }
            },
            hir::ExprKind::Continue(dest) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Continue {name: 'Continue', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
```

```
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                if let Ok(target) = dest.target_id {
                    self.create_link(ex.hir_id, "Jump", 0, target);
                }
            },
            hir::ExprKind::Ret(value) => {
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Return {name: ’Return’, hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                if let Some(value) = value {
                    let child_hir_id = self.visit_expr(value);
                    self.create_link(ex.hir_id, "Child", 0, child_hir_id);
                }
            },
            hir::ExprKind::Lit(lit) => {
                let value = format!("{:?}", lit.node);
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Lit {name: ’Lit’, hir_id: {hir_id}, from_expansion: {from_expansion}, value: {value}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion,
                            "value" => &value
                        }
                    )
                    .unwrap(),
                );
            },
        }

        ex.hir_id
    }

    fn visit_block(&mut self, block: &’tcx hir::Block<’tcx>) -> hir::HirId {
        let from_expansion = block.span.from_expansion();
        let hir_id = serialize_hir_id(block.hir_id);

        self.query.push(
            cypher_stmt!(
                "CREATE (:Block {name: ’Block’, hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                    "hir_id" => &hir_id,
                    "from_expansion" => from_expansion
                }
            )
            .unwrap(),
        );

        for (index, stmt) in block.stmts.iter().enumerate() {
```

```rust
            let child_hir_id = self.visit_stmt(stmt);
            self.create_link(block.hir_id, "Child", index, child_hir_id);
        }

        if let Some(value) = block.expr {
            let child_hir_id = self.visit_expr(value);
            self.create_link(block.hir_id, "Expr", 0, child_hir_id);
        }

        block.hir_id
    }

    fn visit_stmt(&mut self, stmt: &'tcx hir::Stmt<'tcx>) -> hir::HirId {
        match stmt.kind {
            hir::StmtKind::Local(local) => {
                let from_expansion = stmt.span.from_expansion();
                let hir_id = serialize_hir_id(local.hir_id);
                self.query.push(
                    cypher_stmt!(
                        "CREATE (:Local {name: 'Local', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                            "hir_id" => &hir_id,
                            "from_expansion" => from_expansion
                        }
                    )
                    .unwrap(),
                );

                let pat_hir_id = self.visit_pat(local.pat);
                self.create_link(local.hir_id, "Child", 0, pat_hir_id);

                if let Some(value) = local.init {
                    let child_hir_id = self.visit_expr(value);
                    self.create_link(local.hir_id, "Child", 1, child_hir_id);
                }

                local.hir_id
            },
            hir::StmtKind::Expr(expr) | hir::StmtKind::Semi(expr) => {
                self.visit_expr(expr)
            },
            hir::StmtKind::Item(_item_id) => unimplemented!(),
        }
    }

    fn visit_pat(&mut self, pat: &'tcx hir::Pat<'tcx>) -> hir::HirId {
        let from_expansion = pat.span.from_expansion();
        let hir_id = serialize_hir_id(pat.hir_id);
        self.query.push(
            cypher_stmt!(
                "CREATE (:Pat {name: 'Pat', hir_id: {hir_id}, from_expansion: {from_expansion}})", {
                    "hir_id" => &hir_id,
                    "from_expansion" => from_expansion
                }
            )
            .unwrap(),
        );

        pat.hir_id
    }
}
```

The complete source code is also available under: `https://github.com/xFrednet/rust-clippy/tree/20639740` (Accessed: 2021-11-18). This file is located under: `https://github.com/xFrednet/rust-clippy/blob/20639740/clippy_lints/src/graph_query_linter.rs` (Accessed: 2021-11-18)

B Cypher query for the clippy::vec_init_then_push lint

```
MATCH
    (assign {from_expansion: false}),
    (assign)-[:Child {index: 0}]->(var),
    (assign)-[:Child {index: 1}]->(init_call:Call)-[:Child]->(init:Path)
WHERE
    (var.name = "Pat" OR var.name = "Path")
    AND (assign.name = "Local" OR assign.name = "Assign")
    AND (init.path CONTAINS 'new'
        OR init.path CONTAINS 'with_capacity'
        OR init.path CONTAINS 'default')

MATCH
    (scope)-[assign_edge:Child]->(assign),
    (scope)-[next_edge:Child]->(method:MethodCall)
WHERE
    next_edge.index = assign_edge.index + 1
    AND method.ident = "push"

return assign.hir_id, method.hir_id
```

C Rust implementation to process the graph query results

```
// [...] Lines 1 - 14
// Not part of the `clippy::vec_init_then_push` reimplementation

const DB_URL: &str = "http://neo4j:pw-clippy-query@localhost:7474/db/data";
const VEC_INIT_THEN_PUSH_QUERY: &str = r#"
MATCH
    (assign {from_expansion: false}),
    (assign)-[:Child {index: 0}]->(var),
    (assign)-[:Child {index: 1}]->(init_call:Call)-[:Child]->(init:Path)
WHERE
    (var.name = "Pat" OR var.name = "Path")
    AND (assign.name = "Local" OR assign.name = "Assign")
    AND (init.path CONTAINS 'new'
        OR init.path CONTAINS 'with_capacity'
        OR init.path CONTAINS 'default')

MATCH
    (scope)-[assign_edge:Child]->(assign),
    (scope)-[next_edge:Child]->(method:MethodCall)
WHERE
    next_edge.index = assign_edge.index + 1
    AND method.ident = "push"

return assign.hir_id, method.hir_id
"#;
```

```

declare_clippy_lint! {
    /// ### What it does
    ///
    /// ### Why is this bad?
    ///
    /// ### Example
    /// ‘‘‘rust
    /// // example code where clippy issues a warning
    /// ‘‘‘
    /// Use instead:
    /// ‘‘‘rust
    /// // example code which does not raise clippy warning
    /// ‘‘‘
    pub GRAPH_QUERY_LINTER,
    nursery,
    "default lint description"
}

declare_lint_pass!(GraphQueryLinter => [GRAPH_QUERY_LINTER]);

impl LateLintPass<'_> for GraphQueryLinter {
    // [...] Lines 60 - 66
    // Not part of the ‘clippy::vec_init_then_push‘ reimplementation
    fn check_fn(/* [...] */) {
        // [...] Lines 76 - 86
        // Not part of the ‘clippy::vec_init_then_push‘ reimplementation
        exec_query_post(cx)
    }
}

fn exec_query_post(cx: &LateContext<'tcx>) {
    let graph = GraphClient::connect(DB_URL).unwrap();
    let result = match graph.exec(VEC_INIT_THEN_PUSH_QUERY) {
        Ok(result) => result,
        Err(err) => {
            eprintln!("Query Err: {:#?}", err);
            return;
        },
    };

    for row in result.rows() {
        if let Err(err) = process_vec_init_then_push_row(cx, row) {
            eprintln!("Row Err: {:#?}", err);
        }
    }
}

fn process_vec_init_then_push_row(cx: &LateContext<'tcx>, row: Row<'_>) -> Result<(), GraphError> {
    let assign_id = deserialize_hir_id(row.get("assign.hir_id")?);
    let map = &cx.tcx.hir();
    let local_id;
    let init_expr;
    let mut has_let = false;
    let ident_span;
    let err_span;
    match map.get(assign_id) {
        hir::Node::Local(local) if let Some(init) = local.init => {
            local_id = local.pat.hir_id;
            init_expr = init;
            ident_span = local.pat.span;
```

```
            err_span = local.span;
            has_let = true;
        }
        hir::Node::Expr(expr) if let hir::ExprKind::Assign(path, init, _) = &expr.kind => {
            local_id = path_to_local(path).unwrap();
            init_expr = init;
            ident_span = path.span;
            err_span = expr.span;
        }
        node => {
            eprintln!("Unexpected node: {:#?}", node);
            return Ok(());
        }
    }

    // Checking the type
    // A proper lint implementation would go further then just checking the type but a proper
    // query based implementation might handle the graph representation better
    let ty = cx.typeck_results().expr_ty(init_expr);
    if !is_type_diagnostic_item(cx, ty, sym::Vec) {
        return Ok(());
    }

    let method_id = deserialize_hir_id(row.get("method.hir_id")?);
    let push_expr = map.expect_expr(method_id);
    if let hir::ExprKind::MethodCall(_, _, [self_expr, _push_arg], _) = push_expr.kind {
        if path_to_local_id(self_expr, local_id) {
            let mut s = if has_let { String::from("let ") } else { String::new() };
            s.push_str(&snippet(cx, ident_span, ".."));
            s.push_str(" = vec![..];");

            span_lint_and_sugg(
                cx,
                GRAPH_QUERY_LINTER,
                err_span.to(push_expr.span),
                "calls to ‘push‘ immediately after creation",
                "consider using the ‘vec![]‘ macro",
                s,
                Applicability::HasPlaceholders,
            );
        }
    }

    Ok(())
}

// [...] Lines 166 - 749
// Not part of the ‘clippy::vec_init_then_push‘ reimplementation
```

The complete source code is also available under: https://github.com/xFrednet/rust-clippy/tree/20639740 (Accessed: 2021-11-18). This file is located under: https://github.com/xFrednet/rust-clippy/blob/20639740/clippy_lints/src/graph_query_linter.rs (Accessed: 2021-11-18)