Dirac Notation for Quantum State Preparation in Rust

Felipe Tavares[0009−0004−7359−8995]

Pixeldelic Studio, Sydney, Australia
felipe@pixeldelic.studio

Abstract. This paper introduces *Dirac DSL*: a domain specific language and its implementation for the Rust programming language built with procedural macros. Dirac DSL performs compile-time preparation of quantum states with an input in Dirac-like notation. Such state preparation is useful when performing classical simulations of small quantum systems. Dirac DSL provides a few advantages over standard Rust notation when writing classical quantum simulations: a familiar Dirac *bra-ket* syntax available through ergonomic macros, a runtime of $\mathcal{O}(1)$ (no runtime copy) or $\mathcal{O}(n)$ relative to the length of the prepared quantum state tensor (runtime copy), and an extensible compile-time to runtime interface that can translate the compile-time result to arbitrary runtime tensor formats. We perform a brief qualitative analysis of the currently available Rust quantum system simulation libraries and inspect similar projects in other languages to provide some context and understand the state of the art for macro-based DSLs in quantum state manipulation. We then provide a brief introduction to Dirac notation and present the syntax, semantics and implementation details of the `dirac!` and `xdirac!` macros used in the Dirac DSL. Finally, we also perform a benchmark of the implementation. Source code for reproduction is published at `https://github.com/felipetavares/dirac/`.

Keywords: Quantum Simulation · Rust · DSL · Procedural Macro

1 Introduction

Even though a literature review [6] shows that *Python*, *Julia*, *MATLAB*, and *C/C++* are the usual classical languages used in the field of quantum computing simulation, with *Python* libraries such as QuTip [8], Strawberry Fields [10] and TensorFlow Quantum [3] being extremely widespread paired with Jupyter [9] notebooks, *Rust* [18] is still an interesting alternative choice for quantum simulation projects with a large classical stack due to its excellent software engineering qualities: a strong type system, lower likelihood of memory management bugs, embedded hardware support, among others. These qualities were well explored in its use building the Servo web engine [1].

A review of all projects with descriptions or names matching the keyword `quantum` on crates.io [15], the open repository for Rust packages, finds several

quantum simulation and quantum computing crates, the most prominent one being `qoqo` [14]. However, all the available libraries rely on the preparation of quantum states by adapting the usual quantum mechanics *Dirac* notation to operations on state vectors initialised at runtime following standard Rust notation.

As demonstrated by the analysis of *MATLAB* vs. *Python* in [11], where the former has a strong foothold in all areas of engineering due to the familiar notation when dealing with algebraic calculation and the later providers a developer-friendly experience, both aspects are valuable but developer experience is paramount. Hence, when designing scientific domain specific languages, a tight integration with a developer friendly host language can provide the needed developer experience benefits.

We propose introducing a domain specific language to Rust, *Dirac DSL*, balancing developer experience and domain notation by using Rust as a host language while plugging in the quantum mechanics Dirac notation through the use of compile-time Rust macros. This is a similar approach to the one used by the *SQLx* library [13] for integrating *SQL* queries and the Rust language through the `query!` macro.

A similar strategy was adopted by the Julia Quantum project [12] with the `QuDirac.jl` library, which also uses macros to perform computations on quantum states but differently from the approach we propose, and due to the implementation on the Julia language, it focuses on providing a dynamic macro system that is used to compute quantum states at runtime and not on preparing quantum states at compile time.

1.1 Dirac Notation

The Dirac DSL implementation focuses on a subset of the Dirac notation [2] used by most quantum mechanics texts. It features the use of angular brackets $\langle, \rangle$ paired with vertical bars $|$ to represent vectors in $\mathbb{C}^n$ and commonly used operations.

Bra-kets The most important element of Dirac notation is the *ket*, which represents the quantum state of a $\log_2 n$ qubit system or register:

$$|\mathbf{v}\rangle = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Similarly, a *bra* is constructed as:

$$\langle \mathbf{v}| = |\mathbf{v}\rangle^\dagger = \begin{bmatrix} v_1^* & v_2^* & \dots & v_n^* \end{bmatrix}$$

Where $*$ is the complex conjugate operator.

Kronecker Product A n-qubit ket can also be constructed from the n individual states of its qubits when they are separable:

$$|b_1 b_2 \dots b_n\rangle = |b_1\rangle \otimes |b_2\rangle \otimes \cdots \otimes |b_n\rangle$$

Where $|b_i\rangle$ is one of the following well known states:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \quad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \quad |+\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \quad |-\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix}$$

Inner and Outer Products Inner or dot products represent the projection between a *bra* and a *ket* of the same dimension n resulting in a scalar:

$$\langle \boldsymbol{\psi} | \boldsymbol{\varphi} \rangle = \psi_1 \varphi_1 + \psi_2 \varphi_2 + \cdots + \psi_n \varphi_n$$

The outer product is just the tensor product of a *ket* and a *bra*, with dimensions m and n:

$$|\boldsymbol{\varphi}\rangle\langle\boldsymbol{\psi}| = \begin{bmatrix} \varphi_1\psi_1 & \varphi_1\psi_2 & \cdots & \varphi_1\psi_n \\ \varphi_2\psi_1 & \varphi_2\psi_2 & \cdots & \varphi_2\psi_n \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_m\psi_1 & \varphi_m\psi_2 & \cdots & \varphi_m\psi_n \end{bmatrix}$$

Element-wise Operations Element-wise algebraic vector operation on *bra-kets* (addition and subtraction) and scalar operations follow standard algebraic notation:

$$|\boldsymbol{\varphi}\rangle \pm |\boldsymbol{\psi}\rangle \quad \frac{|\boldsymbol{\psi}\rangle}{c} \quad c|\boldsymbol{\psi}\rangle \quad c \pm |\boldsymbol{\psi}\rangle$$

Where c is a scalar.

Norm The notation for norms is also standard:

$$\| \, |\boldsymbol{\psi}\rangle \, \|$$

1.2 Function-like Procedural Macros

The Rust language has several types of compile-time *macros*: operations that transform code before compilation. One of those is *function-like* macros [17]: they are applied by emulating the syntax of a function call and are implemented as a mapping function between an input stream of Rust tokens (a `TokenStream`) and an output stream of tokens. The mapping is arbitrary and allows full access to all features of the Rust language in its implementation (see Listing 1).

```rust
#[proc_macro]
pub fn function_like_macro(input: TokenStream) -> TokenStream {
    /* implementation */
}
```

Listing 1: Function-like macros in Rust map from a `TokenStream` to a `TokenStream`.

2 Dirac DSL

The Dirac DSL implemented in this paper is a mapping of Dirac notation into ISO 8859 [7] characters, which are a subset of the characters available for Rust macros [16]. We follow a bottom up approach to defining the DSL grammar informally, then expand to a formal *EBNF* (Extended Backus-Naur Form) grammar for each group of syntactical constructs.

2.1 Atoms

Atoms are used to represent all the building blocks for operations. This includes: *bra-kets*, complex numbers, inner products, outer products, norms, and parenthesised expressions.

Inner and outer products are introduced as atoms even though they : inner products require special syntax since the middle | is shared by both the *bra* and the *ket*, hence they cannot be built out of more basic constructs and must be atoms. In contrast, outer products are syntactically just two atoms put together (a *ket* and a *bra*), but to keep their semantics separate from standard products we match them lower in the grammar tree, as an atom.

The translation of atoms from algebraic Dirac notation is mostly straightforward, with `|`, `<`, and `>` being used as delimiters for *bra-kets* and `()` for parenthesised expressions.

Norms are delimited by a single vertical bar `|` on both sides, differently from the double vertical bars ‖ usually used in standard notation. The reasoning here is twofold: it is more readable since the kerning between two `||` characters when written in mono-space fonts is much larger than in the usual, and when written next to *bras* and *kets*, having more vertical bars makes it easy to conflate them with the bars from the *bras* and *kets* themselves (see Listing 2).

2.2 Complex Numbers

Complex numbers are represented in standard decimal notation, with a real part and optionally and imaginary part represented by an addition suffixed with `i` (see Listing 3).

```
<atom> ::= <complex> | <outer> | <inner> | <bra> | <ket> |
↪  <parenthesised> |
<norm>

<outer> ::= "|" <register-state> ">" "<" <register-state> "|"
<inner> ::= "<" <register-state> "|" <register-state> ">"

<bra> ::= "<" <register-state> "|"
<ket> ::= "|" <register-state> ">"
```

```
<parenthesised> ::= "(" <add> ")"
<norm> ::= "|" <add> "|"

<qubit-state> ::= "0" | "1" | "+" | "-"
<register-state> ::= <qubit-state> {<qubit-state>}
```

Listing 2: Grammar for atoms. See Listings 3, 5 for `<complex>` and `<add>`.

```
<complex> ::= <number> [ "+" <number> "i" ]
<number> ::= <digit> {<digit>} [ "." <digit> {<digit>} ]
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Listing 3: Grammar for complex numbers.

2.3 Unary Operations

There are two kinds of unary operations in Dirac DSL: prefix and postfix. The postfix conjugate transpose operator, usually written as $M^\dagger$ is mapped to `'` here. The usual prefix additive inverse operator (`-`) does not need any changes (see Listing 4).

```
<dagger> ::= <atom> [ "'" ]
<inverse> ::= [ "-" ] <dagger>
```

Listing 4: Unary operators. See Listing 2 for `<atom>`.

2.4 Binary Operations

Binary operations do not require many changes from standard notation, with one noteworthy exception: the Kronecker product operator $\otimes$ is represented as `x`. This is a small deviation from the standard symbol, but it was chosen for its simplicity versus alternatives. One caveat to consider is that the equivalent

Unicode characters `U+2297` and `U+2A02` are not considered to be valid by the Rust tokenizer [16] which is implemented according to the Unicode recommendation for identifiers [5].

Division is also represented with `/` instead of fractions as is common when entering expressions in most modern languages.

The lack of any operators is also acceptable in the DSL grammar, which allows for natural multiplications in algebraic notation, expressed by the optional `<mul-op>` (see Listing 5).

```
<mul-op> ::= "*" | "/" | "x"
<mul> ::= <inverse> {[ <mul-op> ] <inverse>}

<add-op> ::= "+" | "-"
<add> ::= <mul> {<add-op> <mul>}
```

Listing 5: Binary operations grammar. See Listing 4 for `<inverse>`

2.5 Expressions

A full expression is simply an additive expression `<add>` since it is the topmost node in the grammar tree.

2.6 White-space Handling

The EBNF grammar we defined requires the whole expression to be written down without any white space between symbols from start to finish. The actual implemented parser works around this limitation by simply allowing terminals to consume trailing and leading white spaces.

3 Implementation

The `dirac!` procedural macro is used as an entry point to the DSL:

```
let state = dirac!(/* DSL */);
```

Statements in Dirac DSL are passed in to the macro and the result is expected to be the tensor obtained after executing the expression.

There are two implementations of the `dirac!` macro with slightly different use cases: `dirac!` and `xdirac!`. The former produces the final output tensor using only standard Rust types, and as such does not require any custom runtime code. However, it might be cumbersome to use it, given most use cases would require further operations on the output tensor to perform the actual runtime calculations required for the simulation. This implies that the usage would be more similar to:

```
let state = to_runtime_tensor(dirac!(/* DSL */));
```

The `xdirac!` macro (`x` for extensible) solves this issue by allowing the macro user to plug in the conversion function using a custom `trait` instead of inserting it on every macro call.

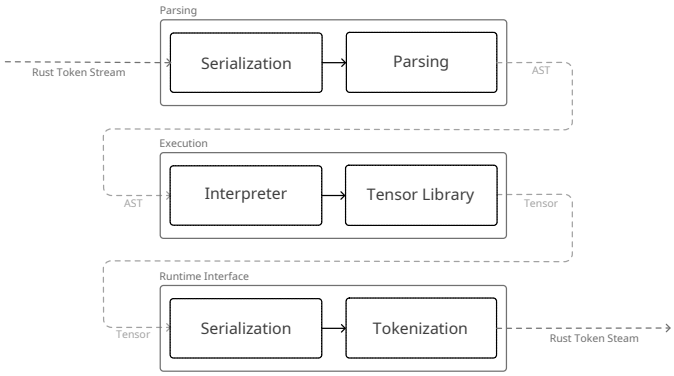

Fig. 1. The `dirac!` macro takes in a stream of tokens and ultimately also produces a stream of tokens. However, the intermediary processing of those streams involve converting to strings, generating a syntax tree and executing the syntax tree to produce a complex vector representing the initialised register state that is finally serialised again.

In Figure 1 we show our approach to the execution of the DSL code inside the `dirac!` macro. We take an interpreter-like approach, where we treat the `dirac!` macro as an interpreter for the DSL. To properly map the DSL into the *function-like* macro paradigm, we implement several intermediary steps to transform both the input and output between the Rust-native `TokenStreams` and other data types more appropriate at each stage of the interpreting process.

1. **Input serialisation** is the first of those transformations: since the tokens in our DSL do not map one to one with tokens in the Rust language, we first de-structure the input into a serialised sequence of bytes.
2. **Parsing** follows, by mapping the the byte sequence into an *AST* (Abstract Syntax Tree).
3. **Execution** then takes the generated AST and interprets it with the help of a tensor computation library, producing a final output tensor.
4. **Runtime interface** again serialises and parses the input back into a Rust stream of tokens.

3.1 Parsing

We utilise the `nom` [4] parsing library to implement our compile-time macros. After converting the input token stream to a string, it is parsed with nom to directly generate the syntax tree of the DSL, instead of relying on the Rust tokenizer or an implementation of thereof such as `syn` [19]. This single-pass architecture allows us to both simplify the implementation and also have control over how individual tokens are defined directly from bytes strings.

The parser implementation uses *combinators*, the `nom` terminology for matching functions capable of either matching parameterised by some input or by other combinators. For example, the `ws` (white space) combinator matches another combinator and leading or trailing white spaces, and hence can help construct white-space-agnostic parsers.

By implementing the language as described in Section 2, the parser outputs a syntax tree defined a recursive `enum`:

```
enum Expression {
    Scalar(Complex64),

    Bra(Tensor),
    Ket(Tensor),

    AdditiveInverse(Box<Expression>),
    Dagger(Box<Expression>),

    Mul(Box<Expression>, Box<Expression>),
    Div(Box<Expression>, Box<Expression>),
    Add(Box<Expression>, Box<Expression>),
    Sub(Box<Expression>, Box<Expression>),
    Kronecker(Box<Expression>, Box<Expression>),

    Inner(Box<Expression>, Box<Expression>),
    Outer(Tensor, Tensor),

    Parenthesised(Box<Expression>),
    Norm(Box<Expression>),
}
```

Every language construct is mapped to one or more `enum` variants, which allows the interpreter to be implemented by recursively pattern matching on `Expression`.

3.2 Interpreting

There are two components to the language interpreter: the interpreter evaluation function itself and the supporting tensor library which executes the computations.

Tensor Computations The supporting tensor library is focused specifically on supporting the kinds of operations needed for the language, as defined in Subsection 1.1. It is implemented as a `Tensor` type with a suite of `impl Tensor` functions and a few trait implementations for common operations where it maps well to Rust's standard operator traits.

The `Tensor` type is implemented as a `struct`:

```
struct Tensor {
    data: Vec<Complex64>,
    shape: (usize, usize)
}
```

Interpreter The interpreter is implemented as a function in `impl Expression`, where `Expression` is the expression enumeration defined in Subsection 3.1. It maps each of the variants into an operation for the supporting tensor library:

1. `Scalar` maps to a rank-0 tensor.
2. `Bra` maps to the conjugate transpose of its inner tensor.
3. `Ket` maps to its inner tensor.
4. `Outer` maps to the outer product of its inner tensors.
5. All other operations map to their recursively evaluated inner expressions and execute the operation afterwards.

3.3 Interfacing Compile-time and Runtime

The final output from the `dirac!` macro is a literal representation of the computed state tensor. It is defined a s fully static type built directly from Rust's native types. This allows the runtime to utilise the output tensor data without needing to include any runtime libraries:

```
type StaticTensor = ((usize, usize), &'static[(f64, f64)]);
```

In most cases we want to perform some computation on this state, with the help of some runtime tensor library. To support this use case, the `xdirac!` macro outputs the same static literal representation but also calls the `to_tensor()` function on it. This will cause an error at runtime because `StaticTensor` is a tuple type that does not implement any traits containing the `to_tensor()` function.

However, if the runtime does implement a trait with the `to_tensor()` function, it can use that to automatically convert the result of all `xdirac!` calls into any type, as defined by the trait.

```
trait ToTensor {
  fn to_tensor(&self) -> RuntimeTensorType;
}

impl ToTensor for StaticTensor { /* ... */ }
```

This enables the runtime to support virtually any tensor library with minimal effort when using the `xdirac!` macro.

The standard `std::convert::From` trait was also considered as an extensibility option, but it would require the tensor libraries themselves to implement the trait, which is not desired since it would only allow the macro to be used with a pre-selected set of tensor libraries that do implement it.

4 Performance

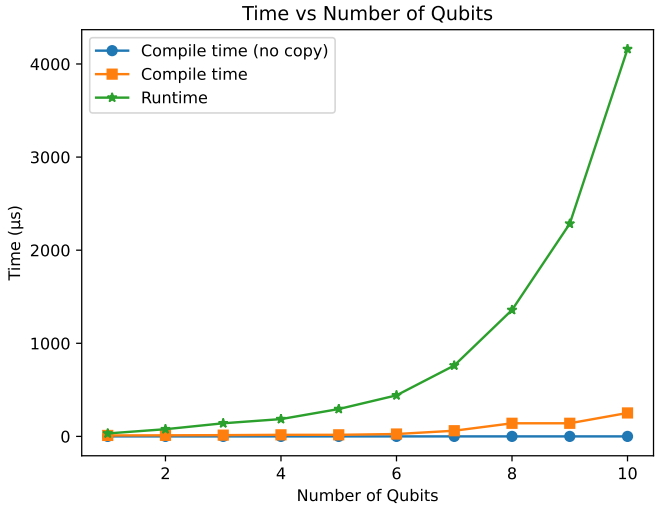

Fig. 2. Calculating the $|0\dots\rangle$ state for n qubits with `dirac!`, `xdirac!` and at runtime.

One of the advantages of using compile-time macros to prepare quantum states is that it allows the pre-computation of those states, reducing run time by caching those values directly in the binary generated by the Rust compiler.

To measure the impact caching has in state preparation, we ran a series of experiments calculating the Kronecker product of an increasing number of qubits from 1 to 10 in the $|0\rangle$ state, generating a final n-qubit $|0\dots\rangle$ state. Each experiment runs for 5 seconds continuously and 100 random execution times are sampled and averaged to produce the final runtime.

In Figure 2 we plot the results for the computation of the n-qubit states at runtime, compile time with `xdirac!` and making copies of the generated tensors at runtime and a baseline $\mathcal{O}(1)$ benchmark with `dirac!` without doing any copies.

5 Conclusions

We explored the design and implementation of Dirac DSL, a small domain specific language that closely resembles the Dirac notation widely used in quantum mechanics literature and research. Through this exploration we found which notation constructs are cumbersome to translate to a streamlined syntax and how we can balance readability with ease of use.

We also found that the integration of Dirac DSL with multiple tensor computation libraries during runtime can be made trivial with the intentional use of user defined traits. Additionally, we show the performance improvements of using Dirac DSL can be substantial depending on the complexity and size of the static prepared states.

Hardware acceleration of compile time tensor calculations and benchmarking techniques for compile time macros are both interesting areas for future works. Expanding the language to support arbitrary operators is another unexplored direction.

References

1. Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J., Sapin, S.: Engineering the servo web browser engine using rust. 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C) pp. 81–89 (2016)
2. Banks, T.: Review of linear algebra and dirac notation. Quantum Mechanics: An Introduction (2018)
3. Broughton, M., Verdon, G., Mccourt, T., Martinez, A.J., Yoo, J.H., Isakov, S.V., Massey, P., Niu, M.Y., Halavati, R., Peters, E., Leib, M., Skolik, A., Streif, M., Dollen, D.V., McClean, J.R., Boixo, S., Bacon, D., Ho, A.K., Neven, H., Mohseni, M.: Tensorflow quantum: A software framework for quantum machine learning. ArXiv **abs/2003.02989** (2020)
4. Couprie, G.: Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. 2015 IEEE Security and Privacy Workshops pp. 142–148 (2015)
5. Davis, M., Leroy, R.: Uax #31: Unicode identifier and pattern syntax (2022), `https://www.unicode.org/reports/tr31/tr31-37.html`
6. Garhwal, S., Ghorani, M., Ahmad, A.: Quantum programming language: A systematic review of research topic and top cited languages. Archives of Computational Methods in Engineering **28**, 289 – 310 (2019)
7. ISO: Iec 8859 information processing: 8-bit single-byte graphic coded character sets. ISO/IEC 8859-15: 1999 (1999)
8. Johansson, J.R., Nation, P.D., Nori, F.: Qutip 2: A python framework for the dynamics of open quantum systems. Comput. Phys. Commun. **184**, 1234–1240 (2012)
9. Jupyter Development Team: Project Jupyter (2023), `https://jupyter.org`
10. Killoran, N., Izaac, J.A., Quesada, N., Bergholm, V., Amy, M., Weedbrook, C.: Strawberry fields: A software platform for photonic quantum computing. Quantum (2018)
11. Ozgur, C.O., Colliau, T., Rogers, G., Hughes, Z., Myer-Tyson, E.B.: Matlab vs. python vs. r. Journal of data science **15**, 355–371 (2021)

,

12. The JuliaQuantum Team: Juliaquantum. https://github.com/JuliaQuantum/ (2023), accessed: 2023-03-19
13. The Launchbadge Team: The rust sql toolkit. https://github.com/launchbadge/sqlx (2023)
14. The qoqo Team: qoqo: Quantum operation quantum operation. https://github.com/HQSquantumsimulations/qoqo (2023)
15. The Rust Project Developers: crates.io: Rust package registry. https://crates.io/ (2023), accessed: 2023-03-19
16. The Rust Project Developers: Identifiers - the rust reference (2023), https://doc.rust-lang.org/reference/identifiers.html
17. The Rust Project Developers: Procedural macros - the rust reference (2023), https://doc.rust-lang.org/reference/procedural-macros.html
18. The Rust Project Developers: The rust reference (2023), https://doc.rust-lang.org/reference/
19. Tolnay, D., et al.: syn: Parser for Rust source code (2023), https://github.com/dtolnay/syn