Automatic Parallelisation of Rust Programs at Compile Time

Author

Michael Oultram

1428105

Degree

MSci in Computer Science

24th March 2018

Supervisor

Dr Ian Batten

Institution

School of Computer Science

University of Birmingham

Contents

Chapter 1

Preamble

1.1 Abstract

TODO: **Write Abstract**

All software for this project can be found at `https://github.com/MichaelOultram/Auto_Parallelise`

1.2 Acknowledgements

Rust Compiler (`https://github.com/rust-lang/rust`)

Serde (`https://serde.rs/`): used to convert rust objects into JSON and back again

Chapter 2

Introduction

Processors are being released with more cores. Sequential code cannot take advantage of these new cores. Writing parallel code is more difficult than writing sequential code. Existing sequential code would need to be rewritten to be parallelised. Ideally we want to gain the benefits of parallel code, whilst only having to write easy sequential code.

One solution to this problem is a parallelising compiler. Research done in this field has mostly focused on the C/C++ language although other researches have had success using other languages. Some methods require manual annotation of the source code by the programmer to specify which parts of the program are parallelisable. Others have attempted to automatically detect these areas, but with an unsafe language like C++ it is challenging.

For my project, I focused on the safe language rust. Rust has a unique way of managing memory such that only one thread can access the memory space at once. This is guaranteed at compile time, and should make the process of automatically detecting dependencies much easier. Rust also allows plugins into the compiler (nightly feature only as of writing) which gives modification access to the abstract syntax tree.

2.1 Rust Language Features

Rust is similar to other programming languages such as C++ but it does has some specific features that may not be known to the reader. This section briefly explains features of the language that are used in later sections of the report. If the reader requires more in depth understanding than what is provided, then they should look at the language documentation (*The Rust Book* 2017).

2.1.1 Safety Features

“Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety” (*The Rust Programming Language* 2017). To get these safety properties, rust has some unique features. The biggest difference to other programming languages is how variable are handled.

Ownership

In Rust, all variables have an ownership. Only one block can have access to that variable at a time. This is enforced at compile time.

```
fn main() {
    let a = 10;
    f(&a);
    g(a);
    // Cannot access a here anymore
}
```

```
fn f(a: &u32){} // f borrows a
fn g(a: u32){} // g takes ownership of a
```

Listing 1: Borrowing and moving example

In this example, `a` is a local variable in the `main` method.

When `f` is called with parameter `a`, the function borrows that variable. This is similar to call-by-reference from other programming languages.

When `g` is called with parameter `a`, the variable is moved to `g`. This is unlike other programming languages as this is not call-by-value. Instead `g` takes ownership of `a`. When `g` is returned, the main method can no longer use `a`.

Mutability

Variables mutability is declared when the variable is declared. In rust, variables are immutable by default, but if specified they are mutable. When a variable is borrowed, it can either be immutably borrowed or mutably borrowed.

```
fn main() {
    let a = 10;
    let mut b = 20;
    f(&a, &b);
    g(&mut b);
}
fn f(a: &u32, b: &u32){} // f immutably borrows a and b
fn g(b: &mut u32){} // g mutably borrows b
```

Listing 2: Immutable and mutable borrowing

In the `main` method of this example, `a` is an immutable local variable and `b` is a mutable local variable. The `f` function borrows both `a` and `b` immutably. Even though `b` is declared as mutable, it cannot be changed inside `f`. Once `f` returns, `b` becomes mutable again inside the `main` method. The `g` function shows how `b` can be borrowed mutably.

Unsafe Blocks

The programmer may want can turn off some of rust's safety features by using an unsafe block. The most common use of an unsafe block is to modify a mutable static variable but it also allows de-referencing of a raw pointer and calling unsafe functions (i.e. an external c function). Using unsafe blocks may introduce race conditions as two threads could try to modify a global at the same time, and the rust language would not guarantee an order.

```
static mut global: u32 = 3;
fn main() {
    let a = global;
    inc_global();
    unsafe {
        global = 5;
    }
}

unsafe fn inc_global() {
    global += 1;
}
```

Listing 3: Immutable and mutable borrowing

2.1.2 Threads

Rust uses real threads.

Channels

2.1.3 Crates

2.2 Related Work

2.2.1 Parallelisation Models

In this section, we look at theoretical models of automatic parallelism. The static parallelism subsection shows related work where the schedule is fixed and calculated at 'compile' time. It is shown how rearranging loop iterations and optimising memory access patterns for multiple threads can increase performance. The speculative parallelism subsection shows related work where the schedule is more flexible. This kind of parallelism tries to run dependent tasks in parallel and detecting when there is a conflict. When a conflict occurs, some parallel thread is 'undone' and rerun.

Static parallelism

Feautrier (1992) describes one model of a parallel program as a set of operations Ω on an initial store, and a partial ordering binary relation Γ also known as a dependency tree. It is shown that this basic model of a parallel is equivalent to affine scheduling, where Ω and Γ are described as linear inequalities. Finding a solution where these linear inequalities hold produces a schedule for the program where dependent statements are executed in order. There are some programs where no affine schedule exists. Bondhugula et al. (2008) uses the affine scheduling model on perfectly, and imperfectly nested loops. They describe the transformations needed to minimise the communication between threads, further increasing the performance of the parallelised code.

An alternative method to affine scheduling is iteration space slicing introduced by Pugh and Rosser (1997). "Iteration space slicing takes dependency information as input to find all statement instances from a given loop nest which must be executed to produce the correct result". Pugh and Rosser (1997) shows how this information can be used to transform loops on example programs to produce a real world speedup. Beletska et al. (2011) shows that iteration space slicing extracts more coarse-grained parallelism than affine scheduling.

Speculative parallelism

Zhong et al. (2008) shows that there is some parallelisable parts hidden in loops that affine scheduling and iteration space splicing cannot find. They propose a method that runs future loop iterations in parallel with past loop iterations. If a future loop iteration accesses some shared memory space, and then a past iteration modifies that same location, the future loop iteration is 'undone' and restarted. It is shown that this method increases the amount of the program that is parallelised.

Prabhu, Ramalingam and Vaswani (2010) introduce two new language constructs for C# to allow the programmer to manually specify areas of the program that can be speculatively parallelised.

Yiapanis, Brown and Luján (2016) designs a parallelising compiler which can automatically take advantage of speculative parallelism.

2.2.2 Parallelisation Implementations

In this section we look at: parallelising compilers which focus on parallelising FORTRAN programs; OpenMP which is an model for shared memory programming and parallelising compilers which convert sequential CPU code into parallelised GPU code. Some of these parallelising compilers are based off of models described in subsection 2.2.1.

Eigenmann, Hoeflinger and D. Padua (1998) manually parallelises the PERFECT benchmarks for FORTRAN which are compared with the original versions to calculate the potential speedup of an automatic parallelising compiler. D'Hollander, Zhang and Wang (1998) developed a FORTRAN transformer which reconstructs code using `GOTO` statements so that more parallelisms can be detected. It performs dependency analysis and automatically parallelised loops by splitting the task into jobs. These jobs can be split between networked machines to run more jobs concurrently. Rauchwerger and D. A. Padua (1999) introduce a new language construct for FORTRAN programs which allows for run-time speculative parallelism on for loops. Their implementation parallelises some parts of the PERFECT benchmarks which existing parallelising compilers of the time could not find.

Quiñones et al. (2005) introduce the Mitosis compiler which combines speculation with iteration space slicing. There is always only one non-speculative thread which is seen as the base execution; all other threads are speculative. The Mitosis compiler computes the probability of two iterations conflicting. If this probability is low, and there is a spare thread unit, then the loop iteration is executed in parallel. The non-speculative thread detects any conflicts as it is the only thread that can commit results.

Dagum and Menon (1998) introduces a programming interface for shared memory multiprocessors called OpenMP targeted at FORTRAN, C and C++. The programmer annotates the elements of the program that are parallelisable, which the compiler recognises and performs the optimisation. OpenMP is compared to alterative parallel programming models. Kim et al. (2000) introduces the ICP-PFC compiler for FORTRAN which uses the OpenMP model. All loops in the source code are analysed by calculated a dependency matrix. The compiler automatically adds the relevant OpenMP annotations to the loop. Lam (2011) extends OpenMP using machine learning to automate the parallelisation. The system is trained using a set containing programs already parallelised using OpenMP. The knowledge learned is applied to sequential programs to produce parallelised programs.

A CPUs architecture is typically optimised for latency whereas a GPUs architecture is typically optimised for throughput. This can make GPUs perform much better that CPUs for a certain type of task. Baskaran, Ramanujam and Sadayappan (2010) uses the affine transformation model to convert sequential C code into parallelised CUDA code. `For` loops are tiled for efficient execution on the GPU.

Chapter 3

Design

This chapter covers the original design concepts of my parallelising compiler without realising the full internal details of the compiler. Once I began developing the compiler, I realised some mistakes in the design which I did not fully think through and some adaptations that I had to do due to the structure of the rust compiler. These changes are described in chapter 4. Also due to time constraints, not all the features described in this chapter were implemented.

The rust abstract syntax tree consists of three main types: Blocks, Statements and Expressions. A block contains a list of statements and a statement is a combination of expressions. A block can be represented as an expression (either directly, or inside a loop/if/etc.) which allows for infinite depth in the tree. Variables are a type of expression. My parallelising compiler will focus on statement level parallelisation within a given block.

Parallelising compilers described in the literature (section 2.2) are split into several stages. Similarly, the design of my compiler is in two main stages, the analysis stage and the modification stage. Both these stages contain several steps to achieve their goal. The analysis stage looks at each function in the source code and tries to find parts that could be parallelised. The modification stage takes the parts that can be parallelised and changes the source code so that they run in parallel.

3.1 Analysis Stage

Algorithm 1 Dependency Analysis Algorithm

```
Require: block as a list of Statements
envs = {}
deps = {}
block_env = []        ▷ Statement Environment of the Block when represented as a Statement
for stmt in block do
    stmt_env = Set of variables that stmt uses
    envs.push(stmt_env)
    stmt_deps = {}
    for i = length(stmt_env) −1 to 0 do        ▷ Search backwards for vars in stmt_env
        dep_env = envs[i]
        for all var where (var is in stmt_env) and (var is in dep_env) do
            stmt_deps.push(i)
            stmt_env.remove(var)
    block_env.push(stmt_env)    ▷ Unmatched variables are part of the block's environment
```

Each statement is analysed individually to provide a list of variables that the statement accesses. This list describes the variable dependencies that the statement has, but it does not describe which statements must be executed before the current one for the program to remain correct. To get this information, the algorithm looks at each statement in turn. For each variable that it requires, it looks at the statements before it in the block in reverse order for that variable to be its list of variable dependencies. The first statement found containing this variable as a dependency is added as a statement dependency. Any variables that were not found above the current statement must be defined outside the current block, and so are added as the current blocks dependency. This algorithm is shown in Algorithm 1.

3.2 Modification Stage

The dependency tree provided by the analysis stage shows what statements can be run in parallel. Some of these statements have multiple dependencies, all of which must be met before the statement is run. Each of these dependencies could be in a separate thread, and so some synchronisation technique is needed. The dependency tree is converted into a schedule tree so that we know which statements are run in which threads, and where/when synchronisation is required between threads.

Algorithm 2 Scheduling Algorithm

```
Require: block as a list of Statements
Require: envs as a list Statement Environments
Require: deps as a list Statement Dependencies
schedule_trees = []
for i = 0 in length(block) - 1 do          ▷ Add all independent statements to separate trees
    if length(deps[i]) == 0 then
        schedule_trees.push(("Run", i, []))
while not all Statements from block are in schedule_trees do
    for stmtid = 0 to length(blocks) - 1 where block[stmtid] is not in schedule_trees do
        if all deps[stmtid] are in schedule_trees then
            dep_trees = find all ("Run", i, _) in schedule_trees for all i in deps[stmtid]
            sort descending dep_trees and deps[stmtid] by depth in schedule_trees
            for i = 1 to length(dep_trees) do
                (_, _, subtrees) = dep_trees[i]
                subtrees.push(("SyncTo", stmtid))
            (_, _, subtrees) = deps[0]
            subtrees.push(("SyncFrom", deps[stmtid][1 :], [("Run", stmtid, [])]))
```

The scheduling algorithm designed aims to run as much as possible in separate threads. It makes the naive assumption that threads have no overhead. To start of with, the algorithm looks for any statements with no dependencies. Each of these statements can be run in a separate thread. For all the remaining statements, the algorithm looks selects the set of statements that have all their dependencies in the schedule already. If the statement only has one dependency, then that statement is added directly after that dependency. If the statement has more than one, then the algorithm looks to see which dependency has the longest chain. The thought behind this is that this dependency should be the slowest, and so all the other dependencies should have been finished by this point. To make sure that there are no race conditions, a syncline is introduced from the other dependencies to just before the current statement. This algorithm is repeated until all the statements are in the schedule. Since there cannot be any cyclic dependencies due to the way that the dependency analysis algorithm was designed, this is guaranteed to terminate. This algorithm is shown in Algorithm 2.

Chapter 4

Implementation

TODO: Explain rust compiler plugin types. Why we compile twice, etc. TODO: Justify all decisions. Explain alternatives considered/implemented and why the design changed

This chapter looks at how the design was implemented in practice and the design decisions that had to be adapted due to unforeseen complexities. **TODO: pad**

There was three choices on how I could implement the design: directly modifying the rust compiler source code and recompiling the compiler; using the rust compiler plugin system to modify the abstract syntax tree or writing a source to source translation from scratch. Modifying the rust compiler would give me the flexibility to change any part of the compiler that I needed but it would make seeing my individual contributions difficult. Also, the compiler itself is very large and complex; it would take a while to compile from a clean state. Using a rust compiler plugin would give me less access, but I would only need to compile my plugin. The complexity of the compiler is still there with this option. Writing a source to source translation system from scratch would allow me to avoid touching the rust compiler and it's complexity. In return, I would have to write code to extract the abstract syntax tree from a source file. I would have to model the ownership/borrowing information to detect when parallelisms properly. From all these choices, I decided to write a rust compiler plugin as it provides all the ownership information is already accessibly. This option will require me to use the full abstract syntax tree; the other options had the possibility of using the less verbose high-level intermediate representation (basically syntax sugar free rust code).

The rust compiler allows for plugins of different types. The two types of plugins used are Syntax Extension plugin and a Linter Plugin. Syntax Extension plugins are run first, and are generally used to convert macros into code. I will use a syntax extension to change the entire body of a function from sequential code into parallelised code. Linter plugins run after all the syntax extension plugins, and are generally used to check code style to produce warnings (like unused variable).

The linter plugin will have the abstract syntax tree of the code with all the macros expanded. This stage has all the information required about dependencies. However, once the rust compiler gets to the linter plugins, I can no longer edit the code (without recompiling the compiler). The solution I decided on was to compile the program twice. On the first compile, the syntax extension does nothing and the linter plugin examines the expanded code. The dependency information gathered is saved into a file for the next compile. On the second compile, the syntax extension plugin reads the file to get all the dependency information. Any parallelisable parts are modified to be run in parallel.

The rust plugin system requires an annotation to be able to access that element of the code. Each function of the sequential source code should be annotated with `#[autoparallelise]`. The AST is described by three types of structs: Block, Stmt and Expr. A block contains a list of statements, and each statement is a combination of expressions. Macros can take arguments and are transformed into code. This transformation happens in the compiler after executing all the syntax extension plugins. Each function is evaluated separately.

4.1 Dependency Analysis

The dependency analysis stage is split over two compiles. The first compile uses the linter plugin, and the second uses the syntax extension plugin.

4.1.1 Linter Plugin

The block of the sequential function is examined statement by statement. Each statement is converted into our representation of the AST so that information can be stored about the dependencies. The extra information includes two environments containing which variables the statement requires and produces.

Once all the statements for a function are converted into our representation, the dependency environment need to be matched up to statement ids. Each converted statement is looked at in turn, starting from the beginning of the function. The algorithm looks backwards from the current point in the function to find what statements produce the variables in the requires environment. The statement ID relative to the block of all of the dependencies is stored as part of the converted statement.

It is possible for a block to be inside a block by being represented as a Statement. Blocks need to be evaluated using the same method as explained in the previous paragraph. A block statement is represented as a list of converted statements, as well as the blocks own environment and dependency ids. It is also possible for a new block to be part of another expression (i.e. for loops). In this case, this is stored as an ExprBlock with the statement that the block originates from, a subtree containing a Block statement (there could be more than one block) and the environments/dependency ids.

Once all functions have been analysed, the DependencyTree is converted into an EncodedDependencyTree. The code part of the converted statement is converted into a statement id. The statement ID is represented as a pair of numbers `(span.lo().0, span.hi().0)`, which relate to the byte location of the source code. This will remain consistent between compile runs, whereas the NodeID does not. All EncodedDependencyTrees and function meta data is stored into a JSON file using `serde_json`.

4.1.2 Syntax Extension Plugin

The plugin detects the JSON file and loads it. This is stored as a shared state between different functions.

The first part of the dependency analysis is repeated for the syntax extension plugin this time. In later section of the design, we need access to the pure AST. There is no (easy) way for the AST to be store into a JSON file and recreated into structs that I could find. The reason that we use the linter plugin is so that we can see the dependencies hidden inside macros.

The dependencies are merged function by function from the shared state so that unexpanded macros get the missing dependencies. The dependencies of Statements that have the same StmtID are merged together.

4.2 Scheduler

Once the dependency analysis is complete, the scheduler takes the dependency tree and works out a schedule. All relative dependency ids are converted into StmtIDs. The idea around the scheduling algorithm is Maximum Spanning Trees. All statements that have no dependencies can be started at the very beginning. All remaining statements wait for all their dependencies to be put into the schedule. Once all the dependencies for a statement are added, this statement is selected as the next one to add to the schedule. As each statement requires all of its dependencies before it can be executed, it should be scheduled to run after the slowest dependency. This should minimise the amount of time that the statement has to wait for its dependencies. Synclines are created for all the remaining dependencies so that all dependencies are met. Each block gets its own schedule.

4.3 Reconstructor

All parts of the reconstructor algorithm takes part in the second compile.

TODO: Remove

4.4 Sample code section

Here is some text above

rust/src/libsyntax/visit.rs

```
pub enum FnKind<'a> {
    /// fn foo() or extern "Abi" fn foo()
    ItemFn(Ident, Unsafety, Spanned<Constness>, Abi, &'a Visibility, &'a Block),

    /// fn foo(&self)
    Method(Ident, &'a MethodSig, Option<&'a Visibility>, &'a Block),

    /// |x, y| body
    Closure(&'a Expr),
}
```

Listing 4

rust/src/libsyntax/ast.rs

```
pub struct Block {
    /// Statements in a block
    pub stmts: Vec<Stmt>,
    pub id: NodeId,
    /// Distinguishes between `unsafe { ... }` and `{ ... }`
    pub rules: BlockCheckMode,
    pub span: Span,
    pub recovered: bool,
}
```

Listing 5

rust/src/libsyntax/ast.rs

```
pub struct Stmt {
    pub id: NodeId,
    pub node: StmtKind,
    pub span: Span,
}
```

Listing 6

rust/src/libsyntax/ast.rs

```
pub enum StmtKind {
    /// A local (let) binding.
    Local(P<Local>),

    /// An item definition.
    Item(P<Item>),

    /// Expr without trailing semi-colon.
    Expr(P<Expr>),
    /// Expr with a trailing semi-colon.
    Semi(P<Expr>),
    /// Macro.
    Mac(P<(Mac, MacStmtStyle, ThinVec<Attribute>)>),
}
```

Listing 7

```
let a; // Local without init
a = {
    let b = vec![1,2,3]; // Local with init
    println!("{:?}", b); // Mac
    b.len() // Expr
}; // Semi
```

Listing 8: Example showing different StmtKinds

rust/src/libsyntax/ast.rs

```
pub enum ExprKind {
    /// A `box x` expression.
    Box(P<Expr>),
    /// First expr is the place; second expr is the value.
    InPlace(P<Expr>, P<Expr>),
    /// An array (`[a, b, c, d]`)
    Array(Vec<P<Expr>>),
    /// A function call
    ///
    /// The first field resolves to the function itself,
    /// and the second field is the list of arguments.
    /// This also represents calling the constructor of
    /// tuple-like ADTs such as tuple structs and enum variants.
    Call(P<Expr>, Vec<P<Expr>>),
    /// A method call (`x.foo::<'static, Bar, Baz>(a, b, c, d)`)
    ///
    /// The `PathSegment` represents the method name and its generic arguments
    /// (within the angle brackets).
    /// The first element of the vector of `Expr`s is the expression that evaluates
    /// to the object on which the method is being called on (the receiver),
    /// and the remaining elements are the rest of the arguments.
    /// Thus, `x.foo::<Bar, Baz>(a, b, c, d)` is represented as
    /// `ExprKind::MethodCall(PathSegment { foo, [Bar, Baz] }, [x, a, b, c, d])`.
    MethodCall(PathSegment, Vec<P<Expr>>),
    /// A tuple (`(a, b, c ,d)`)
    Tup(Vec<P<Expr>>),
    /// A binary operation (For example: `a + b`, `a * b`)
    Binary(BinOp, P<Expr>, P<Expr>),
    /// A unary operation (For example: `!x`, `*x`)
    Unary(UnOp, P<Expr>),
    /// A literal (For example: `1`, `"foo"`)
    Lit(P<Lit>),
    /// A cast (`foo as f64`)
    Cast(P<Expr>, P<Ty>),
    Type(P<Expr>, P<Ty>),
    /// An `if` block, with an optional else block
    ///
    /// `if expr { block } else { expr }`
    If(P<Expr>, P<Block>, Option<P<Expr>>),
    /// An `if let` expression with an optional else block
    ///
    /// `if let pat = expr { block } else { expr }`
    ///
```

```
        /// This is desugared to a `match` expression.
        IfLet(P<Pat>, P<Expr>, P<Block>, Option<P<Expr>>),
        /// A while loop, with an optional label
        ///
        /// `'label: while expr { block }`
        While(P<Expr>, P<Block>, Option<SpannedIdent>),
        /// A while-let loop, with an optional label
        ///
        /// `'label: while let pat = expr { block }`
        ///
        /// This is desugared to a combination of `loop` and `match` expressions.
        WhileLet(P<Pat>, P<Expr>, P<Block>, Option<SpannedIdent>),
        /// A for loop, with an optional label
        ///
        /// `'label: for pat in expr { block }`
        ///
        /// This is desugared to a combination of `loop` and `match` expressions.
        ForLoop(P<Pat>, P<Expr>, P<Block>, Option<SpannedIdent>),
        /// Conditionless loop (can be exited with break, continue, or return)
        ///
        /// `'label: loop { block }`
        Loop(P<Block>, Option<SpannedIdent>),
        /// A `match` block.
        Match(P<Expr>, Vec<Arm>),
        /// A closure (for example, `move |a, b, c| a + b + c`)
        ///
        /// The final span is the span of the argument block `|...|`
        Closure(CaptureBy, P<FnDecl>, P<Expr>, Span),
        /// A block (`{ ... }`)
        Block(P<Block>),
        /// A catch block (`catch { ... }`)
        Catch(P<Block>),

        /// An assignment (`a = foo()`)
        Assign(P<Expr>, P<Expr>),
        /// An assignment with an operator
        ///
        /// For example, `a += 1`.
        AssignOp(BinOp, P<Expr>, P<Expr>),
        /// Access of a named struct field (`obj.foo`)
        Field(P<Expr>, SpannedIdent),
        /// Access of an unnamed field of a struct or tuple-struct
        ///
        /// For example, `foo.0`.
        TupField(P<Expr>, Spanned<usize>),
        /// An indexing operation (`foo[2]`)
        Index(P<Expr>, P<Expr>),
        /// A range (`1..2`, `1..`, `..2`, `1...2`, `1...`, `...2`)
        Range(Option<P<Expr>>, Option<P<Expr>>, RangeLimits),

        /// Variable reference, possibly containing `::` and/or type
        /// parameters, e.g. foo::bar::<baz>.
        ///
        /// Optionally "qualified",
        /// E.g. `<Vec<T> as SomeTrait>::SomeType`.
        Path(Option<QSelf>, Path),

        /// A referencing operation (`&a` or `&mut a`)
        AddrOf(Mutability, P<Expr>),
        /// A `break`, with an optional label to break, and an optional expression
        Break(Option<SpannedIdent>, Option<P<Expr>>),
        /// A `continue`, with an optional label
        Continue(Option<SpannedIdent>),
        /// A `return`, with an optional value to be returned
        Ret(Option<P<Expr>>),

        /// Output of the `asm!()` macro
        InlineAsm(P<InlineAsm>),

        /// A macro invocation; pre-expansion
        Mac(Mac),

        /// A struct literal expression.
```

```
    ///
    /// For example, `Foo {x: 1, y: 2}`, or
    /// `Foo {x: 1, .. base}`, where `base` is the `Option<Expr>`.
    Struct(Path, Vec<Field>, Option<P<Expr>>),

    /// An array literal constructed from one repeated element.
    ///
    /// For example, `[1; 5]`. The first expression is the element
    /// to be repeated; the second is the number of times to repeat it.
    Repeat(P<Expr>, P<Expr>),

    /// No-op: used solely so we can pretty-print faithfully
    Paren(P<Expr>),

    /// `expr?`
    Try(P<Expr>),

    /// A `yield`, with an optional value to be yielded
    Yield(Option<P<Expr>>),
}
```

Listing 9

rust/src/libsyntax_pos/lib.rs

```
pub struct SpanData {
    pub lo: BytePos,
    pub hi: BytePos,
    /// Information about where the macro came from, if this piece of
    /// code was created by a macro expansion.
    pub ctxt: SyntaxContext,
}
```

Listing 10

Here is some text below **??**

Chapter 5

Evaluation

5.1 Successful Sequential Programs Parallelised

5.1.1 Simple Example

5.1.2 Password Cracker

5.2 Failed Sequential Programs Parallelised

My implementation to the problem does not work for all sequential programs.

Chapter 6

Discussion

TODO: Write Discussion

Chapter 7

Conclusion

The problem that I wanted to tackle was automatically converting sequential source code into a parallelised program. This is a hard problem which is well established in the field of computer science. It was very unlikely that I could further the research in this field with the amount of time I had.

TODO: Write conclusion

Chapter 8

Appendix

8.1 Submission File Structure

TODO: Write section

8.2 Running the Code

Install rustup and rustc 1.25.0-nightly (0c6091fbd 2018-02-04) compiler:
```
curl https://sh.rustup.rs -sSf | sh -s -- -y --default-toolchain nightly-2018-03-05
```

Verify the version is correct using:
```
rustc --version
```

Create a new crate and write sequential code:
```
cargo init
```

Under `[dependencies]` in `Cargo.toml`, add:
```
auto_parallelise={version="*", git="https://github.com/MichaelOultram/Auto_Parallelise/"}
```

At the top of your `lib.rs` or `main.rs` file, add:
```
#![feature(plugin)]
#![plugin(auto_parallelise)]
```

At the top of every function, add:
```
#[autoparallelise]
```

Compile the code once to run the analysis stage:
```
cargo build --release
```

Normally you would compile the code again with the same command to apply the modifications but due to a bug in the rust nightly compiler (#46489), this doesn't work. Instead you must pipe stdout into a file:
```
cargo build --release > parallel_code.rs
```

Normally you would just run the parallelised code but due to the bug, you will need to create a new project and copy `parallel_code.rs` along with any imports. Then you can:
```
cargo run --release
```

8.3 References

Baskaran, Muthu, Jj Ramanujam and P Sadayappan (2010). "Automatic C-to-CUDA code generation for affine programs". In: *Compiler Construction*, pp. 244–263.

Beletska, Anna et al. (2011). "Coarse-grained loop parallelization: Iteration space slicing vs affine transformations". In: *Parallel Computing* 37.8, pp. 479–497.

Bondhugula, Uday et al. (2008). "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model". In: *International Conference on Compiler Construction*, pp. 132–146.

D'Hollander, Erik H, Fubo Zhang and Qi Wang (1998). "The fortran parallel transformer and its programming environment". In: *Information sciences* 106.3-4, pp. 293–317.

Dagum, Leonardo and Ramesh Menon (1998). "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1, pp. 46–55.

Eigenmann, Rudolf, Jay Hoeflinger and David Padua (1998). "On the automatic parallelization of the Perfect Benchmarks (R)". In: *IEEE Transactions on Parallel and Distributed Systems* 9.1, pp. 5–23.

Feautrier, Paul (1992). "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International journal of parallel programming* 21.5, pp. 313–347.

Kim, Hong Soog et al. (2000). "ICU-PFC: An automatic parallelizing compiler". In: *Proceedings - 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, HPC-Asia 2000*, pp. 243–246.

Lam, Nam Quang (2011). "A Machine Learning and Compiler-based Approach to Automatically Parallelize Serial Programs Using OpenMP". In: *Master's Projects* 210.

Prabhu, Prakash, Ganesan Ramalingam and Kapil Vaswani (2010). "Safe programmable speculative parallelism". In: *ACM Sigplan Notices* 45.6, pp. 50–61.

Pugh, William and Evan Rosser (1997). "Iteration space slicing and its application to communication optimization". In: *Proceedings of the 11th international conference on Supercomputing.* ACM, pp. 221–228.

Quiñones, Carlos García et al. (2005). "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices". In: *ACM Sigplan Notices* 40.6, pp. 269–279.

Rauchwerger, Lawrence and David A. Padua (1999). "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization". In: *IEEE Transactions on Parallel and Distributed Systems* 10.2, pp. 160–180.

The Rust Book (2017). URL: `https://doc.rust-lang.org/book/` (visited on 12/11/2017).

The Rust Programming Language (2017). URL: `https://www.rust-lang.org` (visited on 12/11/2017).

Yiapanis, Paraskevas, Gavin Brown and Mikel Luján (2016). "Compiler-Driven Software Speculation for Thread-Level Parallelism". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38.2, pp. 1–45.

Zhong, Hongtao et al. (2008). "Uncovering hidden loop level parallelism in sequential applications". In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 290–301.