Automatic Parallelisation of Rust Programs at Compile Time

Author

Michael Oultram

1428105

Supervisor

Dr Ian Batten

Degree

MSci in Computer Science

September 2014 – June 2018

Institution

School of Computer Science

University of Birmingham

Chapter 1

Preamble

1.1 Contents

1.2 Abstract

TODO: Write Abstract

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

Chapter 2

Introduction

2.1 Related Work

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.

TODO: Mention that it is a well researched area 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.

2.1.1 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.

2.1.2 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 2.1.1.

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.

2.2 My Solution using Rust

Due to the limited time of the project, I am unlikely to contribute new ideas to the field. **TODO: Add link here.** 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.

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).

2.2.1 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.

```
1  fn main() {
2      let a = 10;
3      f(&a);
4      g(a);
5      // Cannot access a here anymore
6  }
7  fn f(a: &u32){} // f borrows a
8  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.

```
1  fn main() {
2      let a = 10;
3      let mut b = 20;
4      f(&a, &b);
5      g(&mut b);
6  }
7  fn f(a: &u32, b: &u32){} // f immutably borrows a and b
8  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.

```
1  static mut global: u32 = 3;
2  fn main() {
3      let a = global;
4      inc_global();
5      unsafe {
6          global = 5;
7      }
8  }
9
10 unsafe fn inc_global() {
11     global += 1;
12 }
```

Listing 3: Immutable and mutable borrowing

2.2.2 Threads

Rust uses real threads. Due to memory design, only one thread can have access to a variable safely at once. Channels are used to communicate between threads and can move a variable from one thread to another. This variable must implement the Send trait.

2.2.3 Crates

TODO: Write subsection

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 2.1) 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

```
1:  envs = {}
2:  deps = {}
3:  block_env = []      ▷ Statement Environment of the Block when represented as a Statement
4:  for stmt in block do
5:      stmt_env = Set of variables that stmt uses
6:      envs.push(stmt_env)
7:      stmt_deps = {}
8:      for i = length(stmt_env) −1 to 0 do          ▷ Search backwards for vars in stmt_env
9:          dep_env = envs[i]
10:         for all var where (var is in stmt_env) and (var is in dep_env) do
11:             stmt_deps.push(i)
12:             stmt_env.remove(var)
13:     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

```
1:  schedule_trees = []
2:  for i = 0 in length(block) - 1 do          ▷ Add all independent statements to separate trees
3:      if length(deps[i]) == 0 then
4:          schedule_trees.push(("Run", i, []))
5:  while not all Statements from block are in schedule_trees do
6:      for stmtid = 0 to length(blocks) - 1 where block[stmtid] is not in schedule_trees do
7:          if all deps[stmtid] are in schedule_trees then
8:              dep_trees = find all ("Run", i, _) in schedule_trees for all i in deps[stmtid]
9:              sort descending dep_trees and deps[stmtid] by depth in schedule_trees
10:             for i = 1 to length(dep_trees) do
11:                 (_, _, subtrees) = dep_trees[i]
12:                 subtrees.push(("SyncTo", stmtid))
13:             (_, _, subtrees) = deps[0]
14:             subtrees.push(("SyncFrom", deps[stmtid][1 :], [("Run", stmtid, [])])
```

TODO: Maximum Spanning Trees. 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

This chapter looks at how the design was implemented in practice and the design decisions that had to be adapted due to unforeseen complexities. Each design change is justified with an example of why the original design fails, and alternatives that were considered.

4.1 How to implement the design

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 (AST) 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 very 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 AST 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 requires me to use the full AST; the other options had the possibility of using the less verbose high-level intermediate representation (HIR).

The rust compiler allows for plugins of different types. The two types of plugins of interest are Syntax Extension plugins and Linter plugins. Syntax Extension plugins are executed first in the rust compiler pipeline, and are generally used to convert macros into code. Linter plugins are run at a later stage, and are generally used to check code style to produce warnings (like unused variable). It has the complete AST of the code with all the macros expanded. All the information required about dependencies is accessible inside a linter plugin. However, once the rust compiler gets to the linter plugins, the AST can no longer be modified. Syntax extension plugins allow modification of the AST, but they do not have macros expanded. Some dependencies could be missed by trying to analyse the AST at this stage. 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 then modified to be run in parallel.

To use the rust plugin system, the source code will need to be annotated which provides the plugins with access to that element in the AST. Each function of the sequential source code should be annotated with `#[autoparallelise]`. This annotation does not provide any information about the parallelisability of the source code and is purely just a workaround for the rust plugin system.

4.2 Linter Plugin

In the design section of the report, the dependency analysis algorithm takes in a block and examines it statement by statement to extract out the variables that are used. Actually doing this in the compiler plugin required a lot more effort than it seems on the face of it. The compiler calls a function in the plugin for each annotated function. To get the `Block` struct out of this requires expanding the `FnKind` enum. A `Block` contains a list of statements (`Stmt`). The `Stmt` struct contains a `StmtKind` enum. This describes what kind of statement it is, either a local variable binding, an unexpanded macro, or an expression ending with or without a semicolon. For the variable binding, a `Local` struct is given which contains a `Path` representing the variable name. Some variables will be assigned a value at creation and this is represented as an `Expr` struct. Unexpanded macros contain some other types which represent the arguments to the macro, but I did not end up going any deeper into this. Statements which are expression with or without a semicolon will give an `Expr` struct. Similar to the `Stmt` struct, the `Expr` struct contains a `ExprKind` enum which represents the 39 different types of expressions. Most of these options contain another `Expr`, but some contain other types like a `Pat` which is the second way a variable can be represented. Almost all of the different cases in `ExprKind` had to be dealt with to fully explore the tree and extract all the variables a statement uses (its environment).

Each statement is converted into our representation of the AST so that information can be stored about the dependencies. The original design used two types, a expr to represent a single statement and a block to represent a list of exprs and blocks. This design does not deal a `ExprKind` which contains a `Block` properly (i.e. a for loop). My first idea was to split the `Stmt` into a expr and a block and set the block to depend on the expr. This seemed to work for the dependency analysis stage, but not for the reconstruction stage. It was difficult to distinguish which block is the contents of the `ExprKind` if an external block was also dependent on the expr. I created a third type called exprblock which contains one expr and a list of blocks (as if-else statements will require more than one block). Once all the statements for a function are converted into our representation, and each statement had an environment, then the environments could be matched up as described in Algorithm 1. The statement ID stored is relative to the block of all of the dependencies.

Originally only the dependency ids were going to be added to our representation of the AST. Once I got to the scheduler stage, I realised that I needed the environment information for synclines. Later still I realised that just using one environment had a flaw which adds an extra unneeded dependency. If there are two let statement which have the same variable name, the second let statement will be dependent on the first which is not needed. I fixed this by using two environments, one for the variables that the statement depends on, and another environment which lists the variables that the statement releases/produces. As described in section 2.2, a variable's ownership can be moved, and this can be represented by adding it to the requirements environments and not including it in the releases environment. In practice, I used the two environments, but I did not check whether a variable gets moved or not. If the source code provided to my parallelising plugin is correct sequential code, then the variable will not be mentioned in a statement after the point it has been moved.

4.3 Syntax Extension Plugin

The original design did not require two separate compiles. To get the dependency information from the Linter plugin into the Syntax Extension plugin, it was saved to a JSON file using the serde_json library. I could not store the raw AST with my added dependency information as the both the JSON library and the rust compiler are separate crates which I cannot edit. To

combat this, I converted the dependency tree into an encoded dependency tree. Each `Stmt` and `Block` would be represented by a number. These structs already have a `NodeId` element which is what I tried first. Unfortunately, this id was not consistent between compiles. Both of these structs also had a `Span` element which represents the bytes in the source code that the `Stmt` or `Block` represents. Since the source code would not be changed between compiles, this should remain consistent. My StmtID became two numbers, extracted from the AST by accessing `span.lo().0` and `span.hi().0`.

When the Syntax Extension plugin detects the JSON file, it loads the encoded dependency information. There is no easy way to decode the dependency information, so I ran the dependency analysis algorithm again on the AST that the Syntax Extension plugin has access to. This time, macros will not be expanded, and so I had to add a new Mac type to represent this. I did not try to examine the `Macro` as I had the encoded dependency information. I copied the encoded dependency information into the dependency tree with unexpanded macros by matching up the StmtIDs. Later I realised that I should be replacing the dependency information with the encoded dependency information instead of trying to merge it. The reason for this is because the dependency tree with unexpanded macros would have no environment to start of with. When a statement past the macro tries to match up its environment, it will skip checking the macro and match with something above. When the dependencies were merged, the statement would end up depending on two different statements for the same variable (the macro from the expanded dependency tree and then the statement above).

The scheduling algorithm from the design section did not require many changes at all. The only real addition I had to add was an environment to the syncline. This allows the reconstructor to know which variables it should send down the channel, and what variables will be received in the new thread. Once a schedule has being created, it needs to be turned back into code. First I gathered all of the synclines that are in the schedule, and I created a channel for each of them. The channel name was based off of the statement ids of the two points being synced, and the variable names that will be sent down the channel. Each tree in the list of the schedule was fully explored before starting the next tree. They are all given a separate thread, except the last one which uses the current thread. The rust compiler has a few macros for creating new statements which works well for some static statements, and even some statements where a variable name is changed. But in some cases I had to unwrap the entire statement to change one part and then recreate it all the way up again. This was quite annoying as some of the functions or structs I wanted to create were private in the compiler so I had to delve deeper to see what I could make, and work from that. When I managed to create the types, and reconstructed all the code, the compiler would not accept my changes. All the error messages it gave me used the original source code, and were not very helpful. I got the compiler to print out the changes I made as code to stdout, copied it into a separate folder and it compiled just fine. Searching the issues on the rust compiler github, I came across #46489 which seemed to relate to my issue. Since that issue was not fixed by the time I had to demonstrate my project, I ended up writing a bash script to copy the stdout into a separate file and compile for a third time.

TODO: This paragraph is horribly written. Looking at the parallelised source code, I found that sometimes the incorrect value was returned. In most cases, this caused the parallel code to not compile as the return type was incorrect. The order that the parallelised source code was written was all the tree elements in threads were created, then the last element was executed in the current thread. Before the function exits, it joins all of the threads that were created. Most of the time the last element in the schedule is the correct return value, but not always. I reordered the schedule list so that the last thread is always the return value, and I set that block as a variable. The threads are joined and then the return value variable is returned. This method makes the parallelised source code a lot harder to read, but since it does not need to be human readable, I think this solution is ok.

One way that a sequential program can be sped up is by parallelising loops. This type of parallelism was added late into the project, and so I put a few restrictions on it so that I could get it to work in time. These restrictions could be changed to make it work on more types of loops but I focused on for loops where the variable being iterated is just a number. The scheduler does not know the difference between the block of the if statement and the block of a for loop. When it comes the time to put the block back into the statement, the reconstrutor checks what kind of ExprKind it is. If it is a for loop with the restrictions met, it tries to parallelise it. Any external dependencies for the block are passed between iterations using a separate syncline for each variable for each iteration. The algorithm starts from the top of the block looking for each of the external dependencies and places a receive from the last iteration. Then it starts from the bottom looking for the variables and places a send to the next iteration. If one external variable is received at the start of the block and sent at the end of the block then the for loop will be executed sequentially. The block containing the code for an iteration is placed within a thread and the code to create the channels between iterations is placed above it. All of this is placed within the for loop. Above the for loop is where the initial channels are created, and after the for loop is where the external dependencies are sent to the first iteration and received from the last iteration. This approach works for some examples, but as shown in chapter 5, it has many flaws.

Chapter 5

Evaluation

Evaluating the success of my solution is difficult. My implemented solution only works for some examples, as only some features of the language are supported. It focus is on parallelising as much as possible, not necessarily on getting a faster program. **TODO: Fact check this** Many research papers in the field compare their speedup to LLVM, but the rust compiler (like many languages) uses LLVM for the final stages of compilation. Optimisations from LLVM will be applied to both the sequential and parallel versions, and so my results may not be comparable to others.

Two sequential programs were parallelised: a simple example and a password cracker. The dependency tree and schedule are analysed to check the validity of the parallelising compiler. A collection of automatically generated sequential programs were also parallelised and their output was compared with the sequential version to check that the parallelisms have not semantically changed the program. The amount of concurrent threads was recorded, as well as performance of the parallel version is compared with the sequential version to see if the parallelisms have any affect.

5.1 Sequential Programs Parallelised

5.1.1 Simple Example

This simple example is intended as an easy example to understand how the parallelising compiler works. It consists of two independent variables, `a` and `b` which are printed together.

simple-example/main.rs

```
1  #![feature(plugin)]
2  #![plugin(auto_parallelise())]
3
4  #[autoparallelise]
5  fn main() {
6      let mut a = 4;
7      let mut b = 3;
8      a += 1;
9      b += 1;
10     println!("{}, {}", a, b);
11     println!("End of program");
12 }
```

Listing 4: A simple example program

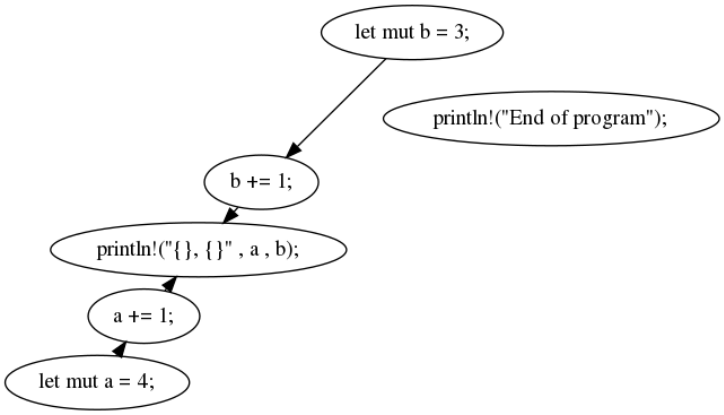

Figure 5.1: Dependency analysis graph of the simple example

The dependency analysis graph clearly shows `a` and `b` do not need to be inter-weaved. It also shows that the `println!(‘‘End of program");` is completely independent of the other print statement. This may not be intended by the programmer, but there is nothing inside the `println!` macro which states that they must be called in order (there is no variable linking the two print statements). This could be fixed by forcing external functions to be executed in order, even if there is no variable linking them together. This would reduce the number of parallelisms detected, but it would make the programs more compatible. Because this dependency is not detected, I sorted the outputs of both sequential and parallel programs so that the order does not matter when comparing outputs.

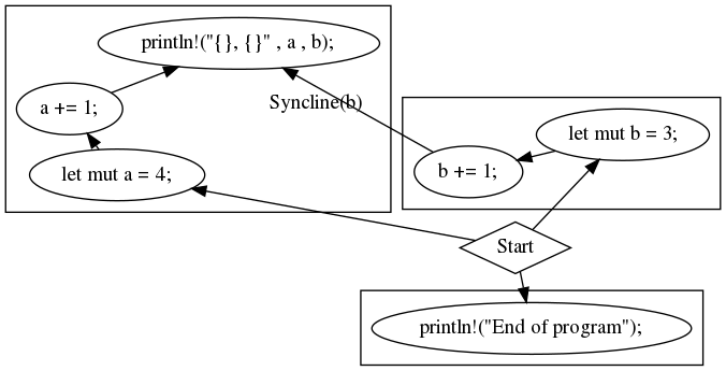

Figure 5.2: Schedule of the simple example

The scheduling algorithm split `a` and `b` into separate threads as well as the “End of program” print statement. The print statement that prints `a` and `b` attached itself to the `a` thread. The `b` variable is sent along a syncline just before the print.

5.1.2 Password Cracker

The password cracker contains three functions: `loadDictionary()`, `hash()` and `main()`. The `loadDictionary()` is not analysed here as the parallelising compiler did not make any changes to this function. This program is on the limits of what my parallelising compiler can do in its current state. Some of the code is written specifically to get it to parallelise properly.

Hash Function

The hash function uses an external crate which implements SHA256. The word argument is iteratively hashed 1000 times and then returned. Notice that `i` is set to decrease from 999 to 0 which does not have any impact on the loop. This has been added to disable the parallelising compiler from trying to parallelise this loop. Line 32 has no dependencies and can be run in parallel. If it was computationally expensive to call `Sha256::new()` then it would make sense to run this statement of the loop in parallel. The rest of the statements depend on the previous iteration and so no speedup can be achieved here. Also note that `hash_word` is borrowed on line 33 which is not a **TODO: is it borrowing, or the str type that breaks this?** fully supported feature of the parallelising compiler. For these reasons, I manually disabled the loop being parallelised in this case. If the performance analyse and borrowing features were implemented, the `.rev()` would not need to be added.

password-cracker/main.rs

```
27  #[autoparallelise]
28  fn hash(word: &String) -> String {
29      let mut hash_word = word.clone();
30      // Hash word using Sha256
31      for i in (0..1000).rev() {
32          let mut hasher = Sha256::new();
33          hasher.input_str(&hash_word);
34          hash_word = hasher.result_str();
35      }
36      hash_word
37  }
```

Listing 5: Hash function of the password cracker program

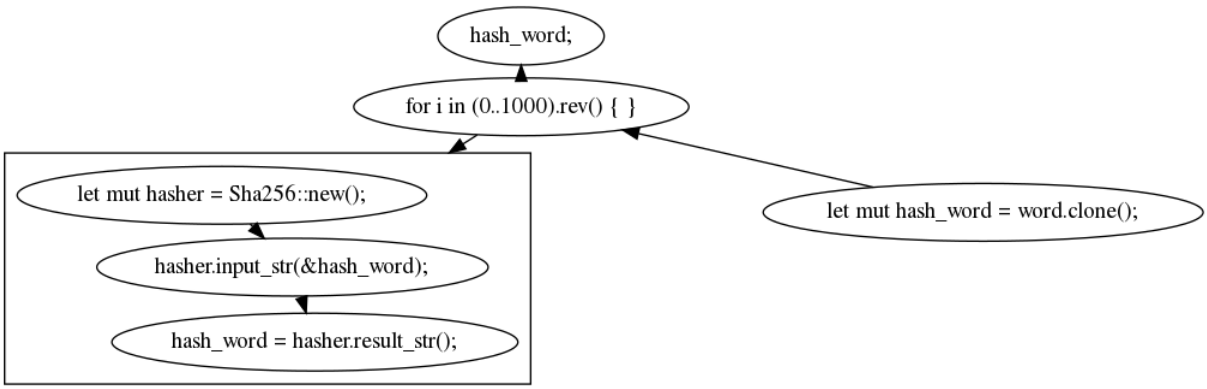

Figure 5.3: Dependency analysis graph of the hash function from the password cracker program

The dependency graph shows how the inner block is linear for one iteration. The reason that the loop cannot be parallelised effectively is that `hash_word` is modified each iteration.

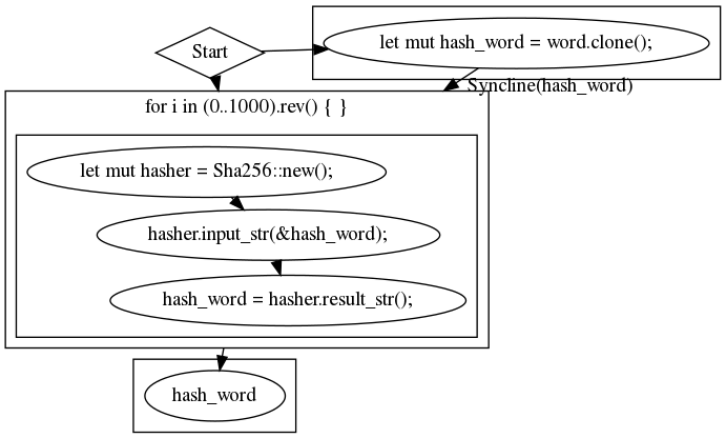

Figure 5.4: Schedule of the hash function from the password cracker program

The schedule shows how linear the iterative hash function is. The only thread separate thread that is spawned clones the word. The reason that the for loop is not inside the same thread is because the `hash_word` dependency is inside the for loop. The condition on the for loop is independent and could be slow and so is parallelised. In reality, this will be much slower due to thread overhead.

Main Method

The main method loads the dictionary from a file into a list of strings. Each word in this list is hashed and compared with a password hash. Notice that the for loop is not interrupted when it finds the correct word, or even stored; it is just printed. A timing circuit is placed around the function to print how long it takes to find the password.

password-cracker/main.rs

```
39  #[autoparallelise]
40  fn main() {
41      let now = Instant::now();
42
43      let dictionary: Vec<String> = load_dictionary();
44      let password_hash =
     ↪   format!("0954229bd82060f9d55ccc310b315ea831b9ba8faee1b76f66a19cc71140dfd7");
45
46      for id in 0..dictionary.len() {
47          let word = dictionary[id].clone();
48          let hash_word = hash(&word);
49          if hash_word == password_hash {
50              println!("Password is {}", word);
51          }
52      }
53
54      let elapsed = now.elapsed();
55      let sec = (elapsed.as_secs() as f64) + (elapsed.subsec_nanos() as f64 / 1000_000_000.0);
56      println!("Seconds: {}", sec);
57  }
```

Listing 6: Main method of the password cracker program

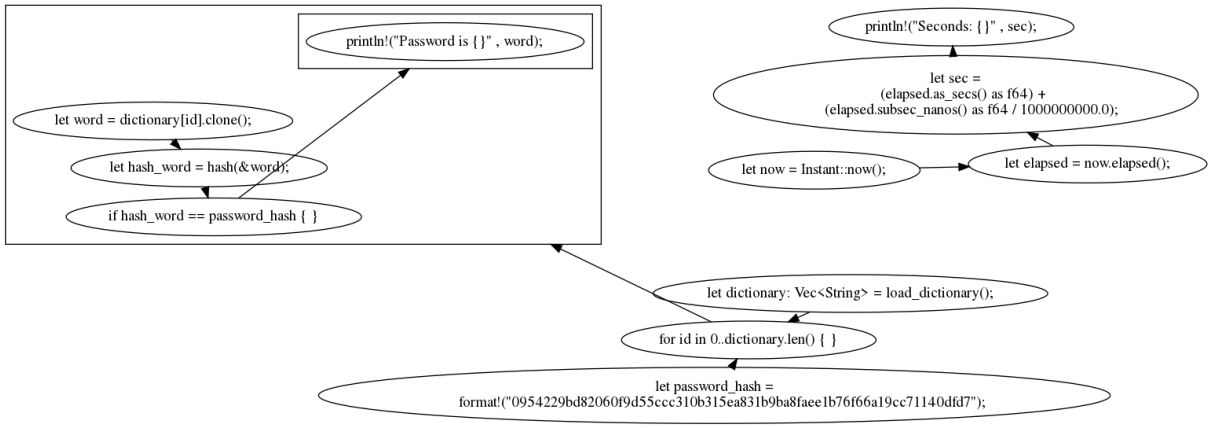

Figure 5.5: Dependency analysis graph of the main method from the password cracker program

As you can see from the dependency graph, the timing circuit is a completely independent program. This is probably not intended, and it is a similar problem to the `println!` problem. There is no real fix I can think of for this issue except hard coding the time crate to be dependent on everything (and everything dependent on it). I am not a fan of hard coding this, as there may be other functions in other crates which also have similar dependency requirements. I will just remove the timing circuit when comparing the outputs as the time will not be consistent between runs.

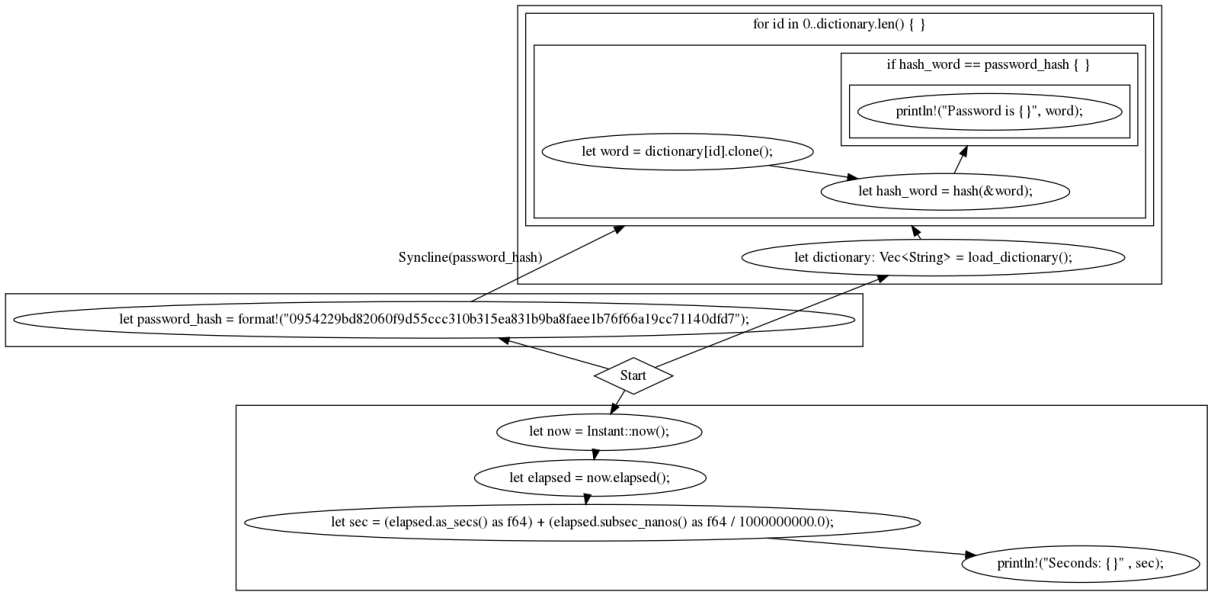

Figure 5.6: Schedule of the main method from the password cracker program

TODO: Show parallelised for loop as a separate schedule

5.1.3 Automatically Generated Sequential Programs

TODO: Write subsection

5.2 Results

TODO: Include results from automated testing

Chapter 6

Discussion

6.1 Achievements

As shown in the evaluation section of the report, my solution manages to separate inter-weaved programs pretty well. It tries to run as much as possible in parallel, and creates syncs for dependencies in different threads. If one statement directly depends on another, then it is placed in the same thread to try to reduce the number of unnecessary threads. Not all programs will compile as not all of the rust language features are implemented properly in the parallelising compiler. One great achievement that I am particularly proud of is the parallel for loops as it was very late into the project before I managed to get them to partially work. Getting the parallelising compiler to work on simple examples took longer than I originally thought, and for a few weeks I had no idea how I was going to parallelise for loops. Once I could parallelise for loops (even partially), I had an example that was not too specifically coded that ran faster parallelised.

6.2 Improvements

The for loop parallelisation could be vastly improved and extended to work with more types of loops. It is not known how many iteration a while loop will execute at compile time and so it is not known how many threads to spawn. The program could spawn one thread at a time and then check if it should terminate, but this would be slower than sequentially running the loop due to thread overhead. To parallelise this kind of loop, some speculative parallelism would have to take place to run faster. The program could spawn 20 threads each running a separate iteration. Any external dependencies should be passed between iterations, as long as the termination condition is not met. If the termination condition is not met by the end of the 20 threads, then it should spawn another batch. When the termination condition is finally met, all the other threads need to be cancelled. In the literature, speculative parallelism requires undoing the cancelled threads, but in this case the cancelled threads are not touching anything external, and so it is as if they have not been run.

More features of the language could be dealt with to parallelise more program, more accurately. Unsafe blocks/functions are not dealt with properly in the current version. They allow for modification of a global variable as well as dereferencing a pure pointer value. Since it would be possible for the unsafe block/function to access a variable without using the variables name, and detecting this would be very difficult, the unsafe block/function should be dependent on all statements before it. That way, the program state in the parallel version should be exactly what the sequential program is. All statements after the unsafe block/function should also be dependent on the unsafe block/function to remain consistent and to stop another statement being scheduled at the same time as the unsafe block/function.

When a variable is sent down a channel, it is moved between threads. The type of the variable must implement the Send trait to allow this to happen. Most types allow for this, but not all types. The current compiler does not check if the type implements this, or even if the variable

is owned and just assumes that it is fine. In the cases that a variable who's type does not implement the Send trait is chosen to be sent down the channel the parallel program will not compile.

The compiler does not check whether a variable is mutable or immutable and assumes all variables are mutable. If the compiler checked this, it may be able to find more parallelisms. Immutable variables are not going to change, and so they could be cloned (if the Clone trait is implemented) into each thread they are used. This would reduce the number of unnecessary syncs, increase the parallel performance. Cloning immutable variables would increase the memory footprint of the runtime parallel program, so some care should be taken to not copy extremely big variables, but in most cases computers have enough memory to cope.

It is important to manage the number of threads being used and the current compiler does not do this. As shown in the evaluation section, when too many threads are spawned, the program does not wait for more threads, it just panics. One technique to combat this is to combine threads together. Not all the threads spawned do enough stuff to overcome the overhead and so the parallel program runs slower. A performance analysis technique would need to be used to estimate whether it is faster of slower to run in a separate thread. If it would be slower, then the scheduler should combine threads to reduce the number of synclines. While this technique will help to combat too many threads, and would solve some of the overhead problems, it might not be enough in some large, computationally expensive parallelisable programs. A threadpool could be used to limit the number of threads that are being executed. Care must be taken to not cause a deadlock using this technique. If all the executing threads are waiting for a dependency in a future thread, then that future thread will not be run as the current threads will never finish.

One last improvement I would suggest to my program is to store the type information with the variable name when deconstructing. Rust has type inference which works most of the time, but as shown in the evaluation section, type inference sometimes fails. By including the type information with the variable, the reconstrutor could annotate each variable with a type in the reconstructed code.

6.3 Approach Changes

While the approach I took managed to find some parallelisms in the code, there are a few changes to the approach that might find more parallelisms more easily. These changes will not be easy to add into my implementation, and so would require starting (essentially) from scratch.

The first approach change I wish to suggest is moving away from statement level parallelisms and into expression level parallelisms. There may be one statement which adds two slow function calls together. If this is written in one statement, it will execute the two slow functions sequentially but expression level parallelisms could spot this. This change significantly increases the difficult of all the stages of the parallel compiler, and increases the chance that the program will return the wrong result.

I would also suggest using the high-level intermediate representation (HIR) instead of using the rust plugins that I used. Some of the code that my parallelising compiler cannot parallelise can be rewritten slightly to make it parallelisable. The main reason for this is there is many ways to write the same program. With a smaller number of expression types, parallelisms can be more focused as the parallelising compiler would not need to deal with as much syntax sugar.

Not all dependencies are directly visible as variables. In the evaluation section, it is shown that a common pattern for timing a function call is completely independent of the function call itself

when looking at variables alone. The programmer intends for the timing circuit to actually time the function, but the parallel code created does not. Other external functions may also have a shared state (unsafe in rust, or an unsafe function call) and could produce other similar problems if the order is changed. The unsafe information may not be visible as if a safe function calls an unsafe function, the safe function remains safe. The parallel compiler would need access to all of the library function body to fully evaluate if it is safe to run in a different order. It may not be possible to detect if the functions can be run at a different time without hard coding.

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. A literature review was included as part of this report, but due to the size of the field, it is no where near complete. It was very unlikely that I could further the research in this field with the amount of time I had.

The solution I chose was to use a compiler plugin which had the downside that I had to deal with all of the syntax sugar. This dramatically increased the number of cases I had to deal with, and not all of them were dealt with correctly.

TODO: Write conclusion

Chapter 8

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.

Chapter 9

Appendix

9.1 Submission File Structure

TODO: Write section

9.2 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`

9.3 Rust Compiler Types

rust/src/libsyntax/visit.rs

```
34  pub enum FnKind<'a> {
35      /// fn foo() or extern "Abi" fn foo()
36      ItemFn(Ident, Unsafety, Spanned<Constness>, Abi, &'a Visibility, &'a Block),
37
38      /// fn foo(&self)
39      Method(Ident, &'a MethodSig, Option<&'a Visibility>, &'a Block),
40
41      /// |x, y| body
42      Closure(&'a Expr),
43  }
```

Listing 7: `FnKind` enum

rust/src/libsyntax/ast.rs

```
489 pub struct Block {
490     /// Statements in a block
491     pub stmts: Vec<Stmt>,
492     pub id: NodeId,
493     /// Distinguishes between `unsafe { ... }` and `{ ... }`
494     pub rules: BlockCheckMode,
495     pub span: Span,
496     pub recovered: bool,
497 }
```

Listing 8: `Block` struct

rust/src/libsyntax/ast.rs

```
781 pub struct Stmt {
782     pub id: NodeId,
783     pub node: StmtKind,
784     pub span: Span,
785 }
```

Listing 9: `Stmt` struct

rust/src/libsyntax/ast.rs

```
815 pub enum StmtKind {
816     /// A local (let) binding.
817     Local(P<Local>),
818
819     /// An item definition.
820     Item(P<Item>),
821
822     /// Expr without trailing semi-colon.
823     Expr(P<Expr>),
824     /// Expr with a trailing semi-colon.
825     Semi(P<Expr>),
826     /// Macro.
827     Mac(P<(Mac, MacStmtStyle, ThinVec<Attribute>)>),
828 }
```

Listing 10: `StmtKind` enum

rust/src/libsyntax/ast.rs

```
899 pub struct Expr {
900     pub id: NodeId,
901     pub node: ExprKind,
902     pub span: Span,
903     pub attrs: ThinVec<Attribute>
904 }
```

Listing 11: `Expr` struct

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 (`{ ... }`)
```

```
1059        Block(P<Block>),
1060        /// A catch block (`catch { ... }`)
1061        Catch(P<Block>),
1062
1063        /// An assignment (`a = foo()`)
1064        Assign(P<Expr>, P<Expr>),
1065        /// An assignment with an operator
1066        ///
1067        /// For example, `a += 1`.
1068        AssignOp(BinOp, P<Expr>, P<Expr>),
1069        /// Access of a named struct field (`obj.foo`)
1070        Field(P<Expr>, SpannedIdent),
1071        /// Access of an unnamed field of a struct or tuple-struct
1072        ///
1073        /// For example, `foo.0`.
1074        TupField(P<Expr>, Spanned<usize>),
1075        /// An indexing operation (`foo[2]`)
1076        Index(P<Expr>, P<Expr>),
1077        /// A range (`1..2`, `1..`, `..2`, `1...2`, `1...`, `...2`)
1078        Range(Option<P<Expr>>, Option<P<Expr>>, RangeLimits),
1079
1080        /// Variable reference, possibly containing `::` and/or type
1081        /// parameters, e.g. foo::bar::<baz>.
1082        ///
1083        /// Optionally "qualified",
1084        /// E.g. `<Vec<T> as SomeTrait>::SomeType`.
1085        Path(Option<QSelf>, Path),
1086
1087        /// A referencing operation (`&a` or `&mut a`)
1088        AddrOf(Mutability, P<Expr>),
1089        /// A `break`, with an optional label to break, and an optional expression
1090        Break(Option<SpannedIdent>, Option<P<Expr>>),
1091        /// A `continue`, with an optional label
1092        Continue(Option<SpannedIdent>),
1093        /// A `return`, with an optional value to be returned
1094        Ret(Option<P<Expr>>),
1095
1096        /// Output of the `asm!()` macro
1097        InlineAsm(P<InlineAsm>),
1098
1099        /// A macro invocation; pre-expansion
1100        Mac(Mac),
1101
1102        /// A struct literal expression.
1103        ///
1104        /// For example, `Foo {x: 1, y: 2}`, or
1105        /// `Foo {x: 1, .. base}`, where `base` is the `Option<Expr>`.
1106        Struct(Path, Vec<Field>, Option<P<Expr>>),
1107
1108        /// An array literal constructed from one repeated element.
1109        ///
1110        /// For example, `[1; 5]`. The first expression is the element
1111        /// to be repeated; the second is the number of times to repeat it.
1112        Repeat(P<Expr>, P<Expr>),
1113
1114        /// No-op: used solely so we can pretty-print faithfully
1115        Paren(P<Expr>),
1116
1117        /// `expr?`
1118        Try(P<Expr>),
1119
1120        /// A `yield`, with an optional value to be yielded
1121        Yield(Option<P<Expr>>),
1122    }
```

Listing 12: `ExprKind` enum

rust/src/libsyntax_pos/lib.rs

```
143 pub struct SpanData {
144     pub lo: BytePos,
145     pub hi: BytePos,
146     /// Information about where the macro came from, if this piece of
```

```
147         /// code was created by a macro expansion.
148         pub ctxt: SyntaxContext,
149     }
```

Listing 13: `SpanData` struct