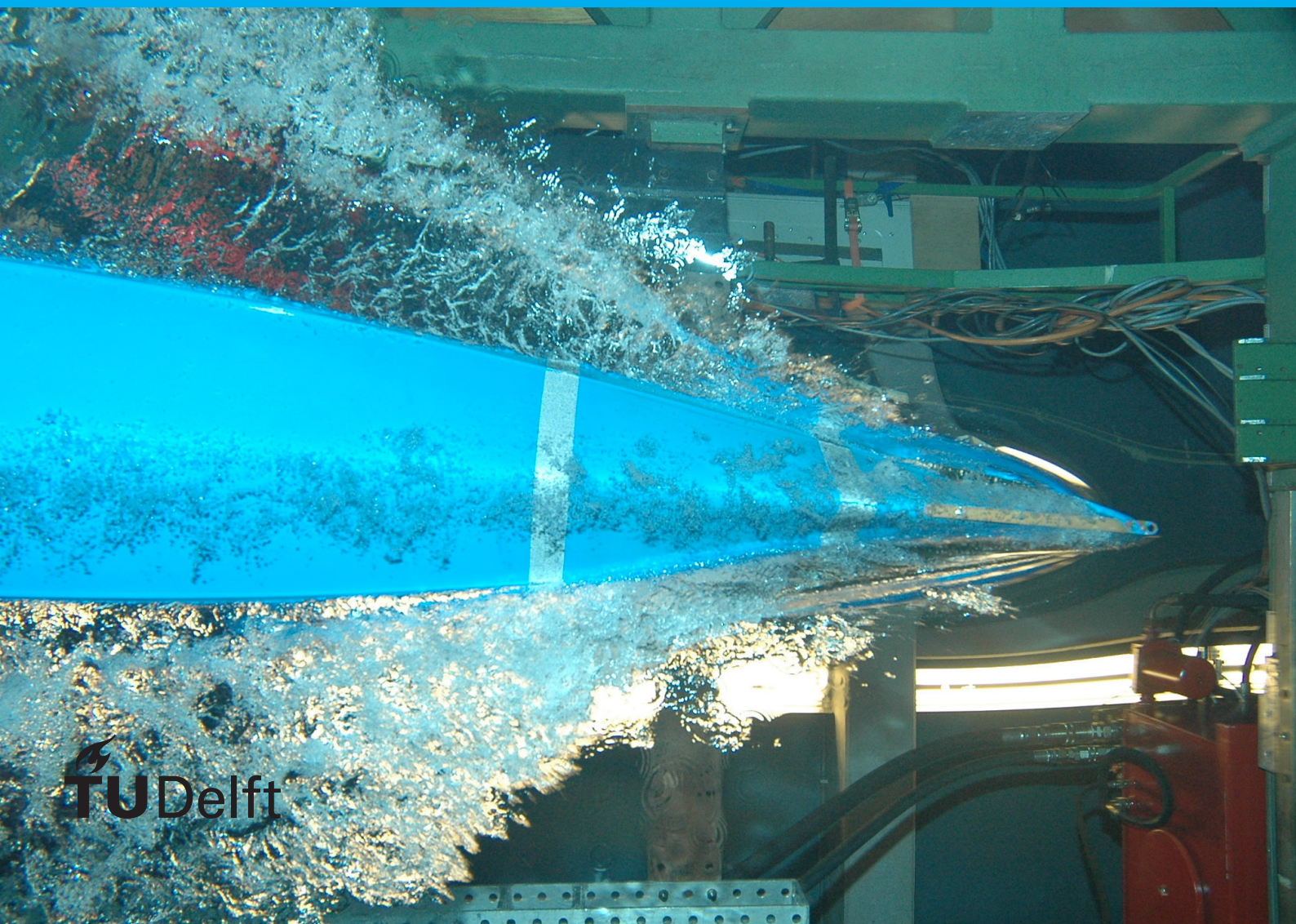
Incremental Type Checking in IncA

Sander Bosma



Incremental Type Checking in IncA

by

Sander Bosma

to obtain the degree of Master of Science in Computer Science
at the faculty EEMCS of the Delft University of Technology,
to be defended publicly on Friday November 2, 2018 at 11:00 AM.

Student number:	4512324	
Thesis committee:	Prof. dr. E. Visser,	TU Delft, chair
	Dr. S. Erdweg,	TU Delft, supervisor
	T. Szabó, MSc,	TU Delft, supervisor
	Dr. A. Katsifodimos,	TU Delft
	Dr. R. Krebbers,	TU Delft

TU Delft

Preface

This thesis has been submitted for the degree of Master of Science in Computer Science at the Delft University of Technology.

When using an integrated development environment, it is desirable to get real-time feedback on the correctness of the program. That is, we want to see the results of the type checker in real-time. However, type checking can take a long time, especially when the subject program is large. To be able to provide real-time results, we need to incrementalize the type checker. This way, when a program changes, we only need to recalculate results for the changed portion of the program, and everything that depends on it. In this thesis, we discuss how we used IncA to implement a type checker for Rust. IncA is a domain specific language for the definition of incremental program analyses: analyses written in IncA are automatically incrementalized. We show in our evaluation that our type checker updates results significantly faster than its non-incremental counterpart. While we were able to successfully implement many of Rust's features, there are some parts we were unable to implement, and we show why that is the case.

I would like to thank Sebastian Erdweg for suggesting the topic of this thesis and for the provided supervision and positivity throughout the project. I would also like to thank Tamás Szabó for his supervision during this project, and for his work on IncA, without which this thesis would not have been possible. Last, but not least, I would like to thank my friends, and especially my family, for their unconditional support.

Sander Bosma
October 26, 2018

Contents

Chapter 1

Introduction

When using an integrated development environment (IDE), it is desirable to obtain feedback regarding the correctness of the code as fast as possible. To provide real-time feedback, the program analysis must be run whenever the code is changed. However, analyzing a complete code base can take a long time when the program is large, especially when the analysis are complex. Fortunately, during development typically only small parts of the program are changed at any time. This usage pattern can be exploited by incrementalizing the analysis. That is, after a change to the code, only the analysis result for the changed code and any code depending on it should be updated. For some analyses, such as the points-to analysis, there are known algorithms that can be implemented in a incremental fashion [16]. However, arguably the most important analysis is the type checker, the result of which not only tells the programmers whether or not the code is valid, but also can be used to help provide code completions. Unfortunately it can be quite difficult to implement a complete incremental type checker.

It is not uncommon for compilers offer some degree incrementality for their type checkers. However, it is often quite coarse-grained; for example, when one node in the AST is changed, they may rerun the complete analysis on the entire file or module. While such incrementality is better than none, a more fine-grained incrementality is required to providing the fastest possible feedback in an IDE.

One interesting recent development in incremental type checking is the concept of co-contextual type-checkers [9]. In a co-contextual type checker the typing rules do not use contexts but instead introduce context *constraints*, which are maintained and solved incrementally. However, co-contextual typing rules can become quite complex [15], and although it has been shown that the incremental performance gain can be significant, achieving that performance requires a lot of manual optimizations. Importantly, this leads to a duplication of effort, since many of these optimizations are language-specific.

In order to reduce duplication of effort, it is desirable to use a language or a framework that automatically incrementalizes the analyses. This way, efforts to increase the performance of the framework are beneficial to all analyses that depends on it. One example of such a language is Datalog, which has known incremental implementations [7, 8]. However, analyses in Datalog are expressed in terms of graph patterns, which many programmer are unfamiliar with and which are not the most intuitive to use. Therefore a higher level language like IncA [26] would be preferred. IncA is a domain specific language (DSL) for program analyses that uses pattern functions to abstract over graph patterns. It has the same expressiveness as Datalog with stratified negation and recursion [26]. Recently, it has been extended [28] to support lattices, which allows both the generation of run-time data and the aggregation of lattice values, making it even more expressive. The incremental performance of analyses written in IncA has been demonstrated on large code bases.

As the target language for the type-checker we chose Rust. It has been chosen because of various reasons. First, it is a relatively modern language that is quickly growing in popularity, making it a relevant real-world language. Second, Rust's type system has some uncommon features that require control-flow sensitive analyses. Finally, at the start of this thesis project, the official compiler (including the type checker) had a reputation to be slow. As such, it is interesting to see if we can make a version of the type checker that is faster for incremental changes than the official compiler.

In this report we document our attempt at using IncA to implement a type checker. The goal of the thesis is two-fold. First, we want to investigate whether IncA is suitable for implementing efficient incremental type checkers. That is, to discover whether it is worthwhile for real-world compilers and

tool chains to start using IncA for the implementation of a type checker. The second goal is to serve as a case-study for Inca; i.e. to explore its expressiveness and to measure the performance of complex analyses. Previously implemented analyses were isolated from each other and relatively small. To implement a type checker for Rust, we need to implement many different typing rules requiring different kinds of analyses, e.g. tree-based type assignment and graph-based (control-flow sensitive) borrow checking. Furthermore, a lot of the analyses are interdependent. It is interesting to see if all rules can be implemented in IncA and to examine the performance of the combined analyses.

Within this thesis, we show that some algorithms that are used in type checking can not be implemented in IncA. For example, the exhaustiveness checking of pattern matches turns out to impossible to implement. The same is true for using unification to implement type inference. We discuss the implementations of various analyses, and see that some analyses can be concisely implemented in IncA. We also show that, due to IncA's limited expressiveness, unfortunately many other analyses need to be implemented in a way that is a lot complex than an equivalent analysis in a general purpose language. Finally, we show IncA is able to effectively incrementalize the implemented analyses.

The rest of this document is structured as follows. Chapter 2 gives a short high-level introduction the Rust language, while Chapter 3 introduces IncA. The next three sections show the implementation of the type checker. Chapter 4 starts with a very small subset of rust. For the most part, each subsection adds some new features to the subset of Rust: they introduce the feature and their typing rules, after which the implementation is discussed. The performance of the implementation is evaluated in Chapter 8, after which in Chapter 9 the related work is examined. Finally, Chapter 10 discusses whether IncA is suitable for implementing type checkers. This section also discusses the difficulties, challenges and common patterns that we encountered while implementing the type checker, and it finishes with a conclusion of the thesis.

Chapter 2

Rust

In this section we give a brief high-level overview of the Rust language, which is the target language of our implemented type checker. While we will introduce the features of Rust in more detail in later sections, it may be useful to have a more high-level understanding of Rust before we dive into the details of specific features in later sections.

Rust is a relatively new language, with version 1 having been released in 2015. It is an imperative language that provides low-level control over memory management, data representation, and concurrency [13]. It aims to be memory safe, concurrent and fast, yet it supports various high-level abstractions, such that the language is not only suitable for applications requiring low-level access to hardware but is suitable for general purpose applications that value safety and speed. Memory safety in Rust is defined to mean that memory that is accessed is guaranteed to be allocated, of the correct type, and, in the case of a read, initialized. Furthermore, heap memory can not accidentally be freed multiple times. Notably, it does not *guarantee* the absence of memory leaks, although it is very difficult to unintentionally leak memory.

Rust distinguishes itself from many other mainstream imperative languages by not supporting inheritance. Instead, Rust relies on the usage of traits, which are similar to interfaces in some languages but unlike interfaces they can be implemented for existing types. Its data types (besides primitives) consists mainly of enums, which are like algebraic data types in many functional languages, and structs. Both can define methods and implement traits.

Rust is not unique in guaranteeing memory safety. In fact, it is quite common for language to provide a memory safety guarantee. However, this guarantee often comes with a cost, either in performance or in expressiveness. For example, it is common for languages to use a garbage collectors and run-time checks to help provide the memory safety. In contrast, no garbage collector and hardly any run-time checks are used in Rust. Instead, it achieves the same by relying on the static guarantees the type checker provides.

Programs that pass the type checker are guaranteed to be free of several types of bugs relating to memory management that are so common in other low-level languages such as C. For example, consider the code in Listing 1, which attempts to create a reference `x` that points to `y`. Since `y` goes out of scope before `x`, this would result in a dangling pointer, which is why Rust's type system prevents this code from compiling. In Rust jargon, we would say that the lifetime of `&y` is too short. In contrast, the equivalent C code, shown in the comments, would compile and run but result in undefined behavior.

```
fn main() {          // void main() {
  let x;             //   bool *x;
  {                  //   {
    let y = true;    //     bool y = true;
    x = &y;          //     x = &y;
  }                  //   }
  let z = *x;        //   bool z = *x;
}                    // }
```

Listing 1: Attempting to dereference a dangling pointer (equivalent C code in comments)

Memory management in Rust uses Resource Acquisition Is Initialization (RAII) that is used in e.g. C++. In particular, when an object goes out of scope, all of its used memory is automatically deallocated. It allows the user to define destructors for objects, although the default implementation should suffice for releasing memory. Supporting RAII does not by itself provide memory safety. The key concept that distinguishes Rust from other languages is that of variable ownership: it uses a type system that is based on the usage of affine types, i.e. it has variables that can be read at most once. This prevents data from being aliased, which ensures that memory can be safely deallocated when a variable goes out of scope. However, since an affine type system by itself is not very ergonomic in use, Rust extends the type system to allow temporary aliasing of data through so-called borrows, which are like pointers but with additional restrictions that ensure that memory safety can be guaranteed. There are two kinds of borrows: mutable borrows can mutate the value they borrow, and immutable borrows can not. The most important restriction is that there can be either a single mutable borrow, or any number of immutable borrows. This allows Rust to guarantee that any well-typed program is free from data races.

Now that the target language of the type checker has been introduced, the next section will introduce IncA, the language we use to implement the type checker.

Chapter 3

IncA

This section introduces IncA. In the next chapters, we discuss how we implemented certain aspects of the type checker in IncA. While these chapters explain most of IncA's features the first time they are used, this section will make later chapter easier to understand.

IncA is a domain specific declarative language for implementing program analyses that are then automatically incrementalized by the runtime system. One of the use cases is the usage inside of integrated development environments (IDEs); inside IDEs it is desirable to get fast feedback after changes to the code. In an IDE, most changes only affect a small portion of the AST. As such, it is often a lot more efficient to update only the analysis result for the changed portion of the AST (and any code that depends on the changed code), rather than rerunning the analysis on the entire AST. In other words, we want analyses to be calculated incrementally. IncA is able to provide such incremental analyses by compiling the program analysis into graph patterns, which are known to support incremental updates. These graph patterns operate directly on the AST.

An analysis in IncA is written in terms of relations, similar to Datalog. These relations are encoded through in functions, which in this document will usually be referred to as *pattern functions*, though sometimes we just call them functions if there is no risk of ambiguity. A pattern function can contain any number of parameters, each of which is an AST node, and return any number of AST nodes, primitives and lattice values. IncA is a declarative language, and thus pattern functions do not describe the control flow. Pattern functions bodies consist of a series of constraints, optionally followed by a return statement, and a result is defined if and only if all the constraints hold.

Two examples of pattern functions are shown in Listing 2. First, consider `invalidArguments`. This pattern function takes an AST node of type `Function` and returns an AST node of type `Argument`. Its purpose is to find arguments which are invalidly named, i.e. to return all arguments that are not uniquely named, or are named "self". Assume that `Function` AST nodes have a list of arguments named `args`, and that `Argument` nodes have a string property called `name`. The first constraint, `arg1 := f.args`, is effectively an existential quantifier; it can be read as $\exists arg1 \in f.args$. In the first two statements, `arg1` and `arg2` are assigned an item from the list `f.args`. The third constraint, `assert arg1 != arg2`, assures that `arg1` and `arg2` are not the same AST node. The properties `arg1.name` and `arg2.name` are of the primitive type `string`, and as such, the constraint `assert arg1.name == arg2.name` checks for string equality rather than AST node equality. For any `arg1` and `arg2` for which all constraints hold, the last statement returns `arg1`. As such, a pattern function can return any number of results for a given combination of arguments. To further clarify the behavior of this pattern function, equivalent imperative pseudocode is shown in the comments. Pattern functions can have any number of bodies; `invalidArguments` actually has a second body, which returns arguments named "self". A function returns all results that are returned by at least one of its bodies.

The other pattern function in Listing 2 only has the constraint `assert undef invalidArguments(f)`. This assertions hold if and only if the call `invalidArguments(f)` returns no results. The inverse of this would be an `assert def` statement. Such assertions are also the only way to use functions that, like `argsOk`, return `Void`. In this case, an `assert def` assertion holds if and only if in at least one bodies, all of the constraints hold. One important thing to note about the usage of `assert undef invalidArguments(f)` is that it requires `invalidArguments` to be completely evaluated before it can be determined whether the constraint holds. If `invalidArguments` were to be dependent on `argsOk`,

either directly or indirectly through other pattern function calls, then IncA can not calculate the pattern function results. As such, IncA will refuse to compile any analysis that can have a call chain cycle if one of the calls is inside an `assert undef` statement.

```
def invalidArguments(f: Function) : Argument = {
  arg1 := f.args                // foreach arg1 in f.args {
  arg2 := f.args                //   foreach arg2 in f.args {
  assert arg1 != arg2           //     assert arg1 != arg2
  assert arg1.name == arg2.name //     assert arg1.name == arg2.name
  return arg1                   //     yield return arg1 }}
} alt {
  arg := f.args
  assert arg.name == "self"
  return arg
}

def argsOk(f: Function) : Void = {
  assert undef invalidArguments(f)
}
```

Listing 2: Example pattern functions

In order to execute an analysis, IncA calculates a computation graph. The first body of `invalidArguments` conceptually consists of three sequential computation nodes:

1. Create a cross product of `f.args * f.args`
2. Filter out tuples where both arguments refer to the same AST node
3. Filter out tuples where the names of both arguments are not the same

Each of these nodes depend on the result of the preceding computation node. Furthermore, node 1 depends on changes in the `args` list inside `Function` AST nodes, and node 3 depends on changes to the `name` property of `Argument` AST nodes. The incrementally in IncA is achieved by only updating a computation node when one of its dependencies has been updated. On other words, computations are updated eagerly. For example, when the name of an argument is changed, only node 3 is being updated. The down side of this is that intermediate results must be stored, resulting increased memory usage. Also, even if there is no call of a pattern function with a certain combination of arguments, the result must still be calculated to support eagerly updating relation. As a result, the initial analysis often takes longer than non-incremental analyses.

One feature that was not shown in the previous example, but that we use a lot, is the switch statement. A switch statement is a constraint that hold if all the constraints in at least one of its bodies holds. An example is shown in Listing 3, with an equivalent program in the comments.

```
def f(e: Expression) : Void = { // def f(e: Expression) : Void = {
  assert e instanceOf VarRef    //   assert e instanceOf VarRef
  switch {                      //   assert def g(e)
    assert def g(e)             // } alt {
  } alt {                       //   assert e instanceOf VarRef
    assert def h(e)             //   assert def h(e)
  }                             // }
}                               //
```

Listing 3: Demonstration of the switch statement

Finally, IncA has support for lattices. These lattices can be used to generate run-time data and add support for aggregating results. They can be used, for example, to denote types. Suppose that

a language has the types `Int` and `Float`, both of which are subtypes of `Number`. The lattice must implement the `glb` and `lub` methods, which return the greatest lower bound and lowest upper bound of two arguments, respectively. A partial, simplified implementation is shown in Listing 4. An example that makes use of this lattice is the `typeOf` pattern function in Listing 5. This function returns the types of both branches of an if-expression. Its return type, `TypeLattice/lub`, indicates that all the results are aggregated using `lub`.

One important limitation of the usage of lattice is that aggregations must be monotonic. As such, it is not allowed to call `lub` aggregations from within and `glb` aggregations, and vise versa.

```
Data Constructors: Int | Float | Number
def lub(t1: TypeLattice, t2: TypeLattice) : TypeLattice {
  if (t1 == t2)
    return t1;
  return TypeLattice.Number;
}
```

Listing 4: Excerpt of lattice implementation

```
def typeOf(e: Expr) : TypeLattice/lub = {
  assert e instanceOf IfExpr
  switch {
    return typeOf(e.ifTrue)
  } alt {
    return typeOf(e.ifFalse)
  }
} alt {
  assert e instance IntExpr
  return TypeLattice.Int
} alt {
  assert e instanceOf FloatExpr
  return TypeLattice.Float
}
```

Listing 5: Lattice usage

IncA currently uses Jetbrains MPS (which is an acronym for Meta Programming System) for developing analyses, and DSLs developed in MPS are the primary target for the analyses. What differentiates MPS from other IDEs is that it uses projectional editing, which means that code is not stored in plain text. Instead of editing a text file, the AST is directly manipulated. When the AST changes, MPS sends a list of the changes to IncA, which IncA uses to update analyses. In MPS, the grammar of a language is defined by describing the structure of separate types of nodes. Nodes can inherit other nodes and implement interfaces. IncA analyses can make use of this type information: the arguments and return types of pattern functions can be interfaces as well. Furthermore, it supports the `assert x instanceOf T` statement to assure that `x` is of type `T`.

This section has given an introduction to IncA. This information on IncA should help the reader to understand the next chapters, which will discuss the implementation of the type checker for Rust that we implemented in IncA.

Chapter 4

Basic type checking

In this section we discuss the type checking of some basic features. We start with a very small subset of Rust, after which each subsection adds some features to the discussed subset.

4.1 Simple types

In this subsection we discuss the type checking of a subset of Rust containing only functions, function calls, boolean primitives, semicolons and blocks.

A function in Rust can have 0 or more named parameters, each of which must have an explicit type annotation. If the function can return a value, the function's return type must be explicitly specified as well. The body of a function consists of a single block. A block is indicated by curly braces, and contains a list of *statements*, which can optionally be followed by a single *expression*. If it does indeed contain an expression `e`, then the block itself is considered an expression, whose value is the value of `e`. Otherwise, it is considered a statement. Expressions can be turned into statements by appending a semicolon.

Variables in Rust are declared by let-bindings. The declared variables are read-only by default; to make them read-write, the `mut` keyword must be added in the declaration. Even though Rust is strongly typed, type inference is used to determine the type of local variables. Thus, although let-bindings can be annotated with a type, this is optional as long as the type can be inferred from its usage. Normal lexical scoping is used: the scope of a variable starts at its declaration and ends when the enclosing block ends. When a new variable is declared and there is already a variable in scope with the same name then the old variable is shadowed by the new one. An example demonstrating the features of this subset of Rust is shown in Listing 6.

```
fn f() {
  h(true);
  g()
}
fn h(arg: bool) { }
fn g() -> bool {
  false
}
```

Listing 6: Feature demonstration

4.1.1 Traditional implementation

The first step that the official Rust compiler takes after parsing code is to transform the Abstract Syntax Tree (AST) into a high-level intermediate representation (HIR). This is essentially a desugaring step that simplifies the set of features the type checker has to deal with. For example, Rust supports three different looping constructs: for-loops, while-loops and unguarded loops; all of these are represented in

the HIR by a single construct (specifically, the loop construct). Additionally, the resolution of variable references is done during the lowering of the AST to the HIR.

Once the HIR has been constructed, the compiler uses [19] Hindley-Milner [23] type inference. In particular, it uses unification. This means that concrete types are not assigned right away. First, each typable node is assigned a type variable. Then, constraints are generated based on the expressions. In the subset of Rust of this subsection, the constraints are very simple. Mainly they would equate `true` and `false` to the boolean type, and it would put constraints on all but the last item of a statement list inside a block to have the unit type. In the complete version of Rust the constraints are a lot more complex. After all constraints are generated they are solved to obtain the concrete types for every expression.

4.1.2 IncA implementation

Unlike the official compiler, the implementation in IncA must be directly done on the AST since IncA does not support program transformations. The structure of the AST is defined in an MPS language. MPS allows the definition of interfaces which AST nodes can implement. All nodes that can be assigned a type implement the `ITypable` interface, including expressions and statements. The goal is to write a pattern function that takes an `ITypable` node and returns its type if it passes the type checker. The first choice we have to make in the implementation of a type checker is how to represent types: we can either represent a type by an AST node, or by a lattice value. For the subsection of Rust discussed in this subsection, it is possible to represent type by AST nodes. The most natural candidate to represent a type is a type annotation AST node, but some programs, such as the example shown in Listing 7, do not have any type annotations. There are other solutions we could try. For example, we could add invisible dummy nodes to the root of the AST to represent Rust's primitive types. However, as we add features such as polymorphic types in the later sections, it becomes impractical to represent types by AST nodes. As such, we choose define a type lattice to represent types. For now, the type lattice contains only the types `Unit` and `Boolean`. The `Unit` type is assigned to functions that do not return values, and to blocks that only contain statements.

```
fn f() {
  true;
}
```

Listing 7: Program without any explicit type annotations

The type checking is done mostly in two phases: the `typeOf` pattern function takes an `ITypable` node as argument returns its type if possible. Afterwards, additional checks are performed by the pattern function `typeOf_withChecks`; it takes the type returned by `typeOf` and only returns it if the node passes all checks. Otherwise no result is returned, signifying that the type is undefined. For example, for a function call, `typeOf` directly returns the declared return type of the called function, while `typeOf_withChecks` checks whether the arguments match in number and type with the function's parameters before it returns the result of `typeOf`. Finally, the entire module is checked using `moduleOk`. This function takes a Module as an argument, which is the root of the AST, and is defined if and only if the module does not contain any items with non-unique names, or nodes that implement the ITypable interface but for which `typeOf_withChecks` does not return a result. We will discuss the implementations of `typeOf` and `typeOf_withChecks` below.

The `typeOf` pattern function is shown in Listing 8. It has one body for each possible input type, where each body starts with an assertion that the argument AST node is of a specific type. Most of these are straightforward: `true` and `false` literals, as well as "bool" type annotations are assigned the type `Boolean`, while semicolons are assigned type `Unit`.

The type of a `Block` AST node depends on whether or not the block contains an expression as its last child. In our AST, `n.expr` has a value if and only if the block contains such an expression. To assign the correct type in both cases, the analysis has two alternatives. The first returns `typeOf(n.expr)`; due to the read of `n.expr`, this alternative returns a result only if `n.expr` has a value. On the other hand, if `n.expr` has no value (i.e. is undefined), then the second alternative returns `Unit`.

The type of a function call expression Depends on the function that is being called. As such, we first need to find this function. The first step in doing so is the statement `assert f instanceOf Function`. This statement introduces the variable `f`, which takes the value of any node in the AST that is of type `Function`. Equivalent imperative pseudocode is shown in the comments for clarity. The next statement `assert f.name == n.functionName` does a string comparison to assure the name of the function matches the name of the function call. As such, the statements in the switch can assume that the call `n` resolves to the function `f`. The remainder of this block has the same structure as the typing of `Block` AST nodes: if the function has defined a return type, then its type is returned; otherwise it defaults to `Unit`.

The remaining interesting case of `typeOf` is for variable references, which is discussed below.

```
def typeOf(n: ITypable): TypeLattice = {
  assert n instanceOf Boolean        // true and false literals
  return TypeLattice.Boolean
} alt {
  assert n instanceOf TypeAnnotation // "bool" in a type annotation
  assert n.name == "bool"
  return TypeLattice.Boolean
} alt {
  assert n instanceOf SemiColon
  return TypeLattice.Unit
} alt{
  assert n instanceOf Block
  switch {
    return typeOf(n.expr)
  } alt {
    assert undef n.expr
    return TypeLattice.Unit
  }
} alt {
  assert n instanceOf FunctionCall // assert n instanceOf FunctionCall
  assert f instanceOf Function     // foreach f of type Function {
  assert f.name == n.functionName  //   assert f.name == n.functionName
  switch {                         //   switch {
    return typeOf(f.returnType)    //     yield return typeOf(f.returnType)
  } alt {                          //   } alt {
    assert undef f.returnType      //     assert undef f.returnType
    return TypeLattice.Unit        //     yield return TypeLattice.Unit
  }                                //   }}
}
```

Listing 8: Implementation of typeOf (part 1)

typeOf_withChecks The implementation of `typeOf_withChecks` to check blocks it quite interesting, since it demonstrates some patterns that are common in IncA analyses. As mentioned, the goal is to check whether the arguments passed in a function call match the function's parameters. We split this up into two separate checks; we need to check whether the number of arguments is correct, and to check whether the types of the arguments match are compatible with the expected types. The code to implement this is shown in Listing 9. Here one body of `typeOf_withChecks` is shown, where we assert that `functionCallArgsNok` must be undefined. This is, we assert that it not the case that the arguments of the call are not correct. While this double negation seems a bit counter intuitive, this is actually the most concise way to implement the check. This is because we want to assert that *all* arguments are correctly typed, and IncA does not provide a built-in way to assert that a function must be defined for all items in a list. This leaves us three options: (1) manually recursing over the items

in the list, (2) using lattice aggregation or (3) using De Morgan's law and instead asserting that the negated assertion does not hold for *any* of the items in the list. Using `undef`, option 3 can be concisely and efficiently implemented, as demonstrated.

Using the functions `resolveCall` to obtain the called function and `CallArgParamPairs` to obtain a pair of an argument in the call and its corresponding parameter in the function, we only need to check whether the types of these are compatible.

IncA recently added the index property. When a node is inside a list, this returns the index of the node in that list. Since it is quite common in type checking to require relating items in one list to items in another list, the index property is very convenient, as can be seen in the implementation of `CallArgParamPairs`. Before IncA supported querying the index, the implementation of `CallArgParamPairs` was a lot less elegant; one body would return a tuple containing the first argument and the first parameter, and another body would do recursion on itself; if `CallArgParamPairs` returned `(a,b)`, then it returns `(a.next, b.next)`.

Finally, `callNumArgsOk` checks whether the number of arguments matches the number of parameters. Since there is no built-in way to obtain the number of items in a list, this needs to be split up into two separate checks; first, this pattern function is defined if the function call has no arguments and the function has no parameters. Second, it is defined if `CallArgParamPairs` returns a pair where both the argument and the parameter are the last elements of their lists. Checking whether an element in the last item of a list is implemented by asserting that its `next` property is undefined. Equivalently we could have compared the indices of the last items of both lists.

```
public def typeOf_withChecks(n : ITypable) : TypeLattice = {
  assert n instanceOf FunctionCall
  assert undef functionCallArgsNok(n)
  return typeOf(n)
} // other bodies omitted
private def functionCallArgsNok(c : Call) : Void = {
  f := resolveCall(c)
  (arg, param) := CallArgParamPairs(c)
  actualTy := typeOf(arg)
  expectedTy := typeOf(param.typeAnnotation)
  assert TypeLattice.leq(actualTy, expectedTy) == false
} alt {
  assert undef callNumArgsOk(c)
}
private def CallArgParamPairs(c : Call) : (Expr, FunctionParam) = {
  f := resolveCall(c)
  arg := c.args
  param := f.params
  assert arg.index == param.index
  return (arg, param)
}
private def callNumArgsOk(c : Call) : Void = {
  (arg, param) := CallArgParamPairs(c)  // more than one arg
  assert undef arg.next
  assert undef param.next
} alt {
  assert c instanceOf FunctionCall      // no args
  f := resolveCall(c)
  assert undef c.arg2
  assert undef f.args
}
```

Listing 9: Checks for function calls

4.2 Name resolution

In this section we extend our subset of Rust with let-bindings, assignments and variable references. Let-bindings are used to declare variables, and they can be given an explicit type annotation. Alternatively, the type annotation can be omitted, to let the type checker infer the type of the variable. Listing 10 shows various examples of the usage of these new features.

```
fn f(arg: bool) {
  let x = true;          // let-binding with inferred type
  let y:bool = x;        // let-binding with explicit type annotation
  let mut x = true;      // shadows the previous declaration of x
  x = false;             // allowed; z resolves to the mutable variable
  let a;                 // delayed initialization
  a = true;
}
fn g() {
  let x = true;
  let x = x; // the x on the right-hand side resolves to the x declared above
}
```

Listing 10: Demonstration of new features in this subset

4.2.1 Implementation

To be able to type variable references, we first need to implement name resolution. Listing 11 shows the analysis we use for the name resolution. The idea behind it is to traverse the AST, starting from the variable reference, until we encounter the let-binding or function parameter that declares the variable. First, we define the order of the AST traversal in the pattern function `nextAstNode`; as its name suggests, it accepts a node and returns the next node to traverse to. This was implemented as follows. The first body returns the `x.prev`, if it exists. That is, if the argument is an item in a list, it returns the previous item in that list. Since `x.prev` could technically be of any type, we assert it to be of the desired type. The second body is a bit more complicated. The basic idea is that if there is no `x.prev`, then we want to return `x.parent`. This is done in the first alternative of the switch. However, we need some additional logic to correctly handle the case shown in `g()` in Listing 10; we need to assure that variable references inside the initialization expressions of let-bindings do not resolve to their enclosing let-binding. This is achieved by the second alternative of the switch; if the parent is a let-binding, we recurse in order to skip that let-binding.

The second step in implementing name resolution is to actually make use of `nextAstNode`. The pattern function `resolveVarRef(n, name)` takes the name of a variable and a node `n` to start the resolution from, and returns the declaration that declares the variable with that name. This function examines the argument `n`; if it declares the variable we are looking from, we are done and return `n`; this is done in the first body of `resolveVarRef`. Otherwise, the variable must have been declared at an earlier part of the code, so we return the result of a recursive call of `resolveVarRef`, where we use the result of `nextAstNode` as the new argument for `n`. This is done in the second body of `resolveVarRef`.

Now that we have implemented name resolution, we can continue with the typing of the new features. First, we will discuss the implementation of the type assignment, i.e. the implementation of `typeOf`. The type assignment for assignments is straightforward: it is always assigned the type `Unit`. This may be surprising, but this is actually the way Rust works: in `a = b = true;`, the variable `a` is indeed assigned the unit type. We also assign `Unit` to let-bindings, since they are statements. The typing of variable references is more complex. Using the pattern function `resolveVarRef` introduced above, a variable reference can be resolved to the let-binding that declares it. However, a type still needs to be assigned to it. It can not simply use the type of the let-binding, since they always have type `Unit()`. Instead, it must use the type of the variable that it declares.

We introduce a new pattern function `typeOfLetVar`, which is shown in Listing 12, that returns the type of the variable that is declared by a let-binding. This pattern function has two bodies: the first

```
def nextAstNode(x : ICFGNode) : ICFGNode = {
  prev := x.prev
  assert prev instanceOf ICFGNode
  return prev
} alt {
  assert undef x.prev
  parent := x.parent
  assert parent instanceOf ICFGNode
  switch {
    assert parent not instanceOf LetBinding
    return parent
  } alt {
    assert parent instanceOf LetBinding
    return nextAstNode(parent)
  }
} alt { /* case for function parameters omitted */ }

def resolveVarRef(n: ICFGNode, name: string) : VarDecl = {
  assert def declaresVar(n, name)
  return n
} alt {
  assert undef declaresVar(n, name)
  return resolveVarRef(nextAstNode(n), name)
}

def declaresVar(n: ICFGNode, name: string) : Void = {
  assert n instanceOf VarDecl
  assert n.name == name
}
```

Listing 11: Name resolution

simply returns `typeOf(l.value)`, which is only successful if `l.value` is defined, i.e. if the let-binding is directly initialized. The second body returns the types assigned to the variable introduces by the letbinding in any assignments. For example, for a let-binding `let x`, it returns the type of `y` for all assignments `x = y` where `x` resolves to the `x` declared by the let-binding. Concretely, the first statement `assert a instanceOf Assignment` introduces a variable `a` which is of the AST node type Assignment. We then assert that the left-hand side of this assignment is a variable reference, and that this variable reference resolves to the variable declared by the argument of `typeOfLetVar`, and then returns its type. The declared return type of `typeOfLetVar` is `TypeLattice/lub`, which means that instead of returning multiple results, it returns the least upper bound of all individually returned values. For the subset of Rust used in this subsection the implementation of the `lub` function is very simple: given types `a` and `b`, it returns `a` if `a` and `b` are equal, and `Invalid` otherwise.

4.3 Structs, enums and methods

In this section we extend our subset of Rust with structs, enums, pattern matches, and methods. Listing 13 shows the syntax used to declare struct and enums. Note that the items use named fields; while Rust additionally supports tuple-like items like `struct Foo(bool)` and unit-like items like `struct Foo;`, we only implement the field-like items, since these are use the most in practice, and the other types do not present additional interesting challenges. The structs we implement are a collection of named fields. Enums are similar to algebraic data types in many functional languages; a value of an enum type can contain one of several variants. In the example, an enum is initialized by the expression `Bar::Variant1{x: true}`. The type of this value is `Bar` rather than `Bar::Variant1`. As such, an

```
private def typeOfLetVar(l : LetBinding) : TypeLattice/lub = {
  return typeOf(l.value)
} alt {
  assert a instanceOf Assignment
  lhs := a.lhs
  assert lhs instanceOf VarRef
  assert resolveVarRef(lhs, lhs.name) == l
  return typeOf(a.rhs)
}
```

Listing 12: Inferring type of a variable from assignments to it

expression `tmp2.x` would be invalid, since the variant is not known during compile time. Likewise, an argument can have type `Bar`, but not `Bar::Variant1`.

```
struct Foo {
  x: bool,
  y: Bar,
}
enum Bar {
  Variant1 {
    x: bool,
  },
  Variant2 {
    x: bool,
    y: bool,
  }
}
fn f(arg1: Foo, arg2: Bar) -> bool {
  let tmp1 = Foo{x: true, y: arg2};
  let tmp2 = tmp1.x;
  let tmp3 = Bar::Variant1{x: true};
}
```

Listing 13: Example struct and enum definitions

To make use of an enum the pattern match expression must be used. An example of this is shown in Listing 14. A pattern match expression can contain one or more match arms, each one of which contains one or more matches. As can be seen, pattern matching is quite powerful. The example shows various aspects of pattern matching; values can match enum variants, structs, literals (e.g. `true`), or wildcards can be used to match any value. Wildcards can be named, in which case they can be referred to from the right-hand side of the match arm, or they can be anonymous (`_`). Match arms can even contain multiple matches, separated by a vertical bar. Interestingly, different patterns in the same match arm can use named wildcards to bind different fields to the same variable; this is shown in the last match arm in the example. When one of the matches in a match arm binds a variable `a`, then all other matches in the same match arm must bind `a` as well, and these must all be of the same type. Furthermore, a match can only bind each variable once, meaning that matches like `Bar::Variant2{x: a, y:a}` to check for equality are illegal. In the full version of Rust, pattern matches must be exhaustive; i.e. all cases must be accounted for. Unfortunately, we can not implement this check in IncA, as we will explain in Section 7.1.

In the previous section let-bindings could only bind a single variable. Now we generalize this by using patterns on the left-hand side of the let-binding. The previous behavior can now be obtained by using a named wildcard for the pattern. The usage of patterns in let-bindings is more limited than inside a pattern match expression: patterns in let-bindings must be irrefutable, such that they are guaranteed to

```
fn is_variant1(arg1: Bar, arg2: Foo) {
  match arg1 {
    Bar::Variant1{x: true} => true,       // matching nested literal
    Bar::Variant1{x: a} => a,             // using the named wildcard a
    Bar::Variant2{x: true, y:_} => true,  // using _, the anonymous wildcard
    // match arm with multiple matches:
    Bar::Variant2{x: true} | Bar::Variant2{x: true, y: true} => true,
    // multiple matches, capturing a different field in both matches
    Bar::Variant2{x: true, y:a} | Bar::Variant1{x: a} => true,
  };
  match arg2 {                            // match on struct rather than enum
    Foo{x:true, y: _} => true,
    _ => false,
  };

  let Foo{x: a, y:b} = arg2;  // bind a to arg2.x and b to arg2.y
}
```

Listing 14: Pattern matching

succeed. For example, a let-binding can not match a variant of an enum. However, it *can* be used to match structs, as the last statement in Listing 14 shows. Parameters in functions now also use patterns, to which the same rules apply as those for let-bindings.

4.3.1 Implementation

The first step in adding support for structs and enums is to determine how to represent them in the type lattice. We chose to represent both structs and enums by `DataType(x)`, where `x` is the AST node of their declaration. This is possible since the type lattice does not need to differentiate between struct and enums. Since these items are not polymorphic, the implementation of `lub` for data types is straightforward. As can be seen in Listing 15, it simply returns one of the arguments if both arguments are equivalent, i.e. if they are both of type `DataType` and refer to the same AST node.

```
def lub(t1: TypeLattice, t2: TypeLattice) : TypeLattice = {
  match (t1, t2) with {
    case (DataType(n1), DataType(n2)) => if (n1 == n2) { return DataType(n1); }
    // omitted: other cases
  };
  return AnyLType;
}
```

Listing 15: Least upper bound for data types

Now that we can represent structs and enums in the type lattice, we need to actually assign this type to nodes. Consider Listing 16; for the struct declaration `struct A`, `typeOf` simply returns `DataType(A)`. For the typing of the argument `arg:A` and the initialization expression `A{}` we just find a struct or enum declaration with the same name, and return the `typeOf` of that declaration.

The typing of field lookups is slightly more difficult. Field lookups have the form `expr.fld`, where `expr` can be any arbitrary expression, e.g. a struct initialization, a function call, or a variable reference. This means that it would be a lot of work to write a pattern function that works on all these different types of expressions. We chose a different approach that is shown in Listing 17. To type `expr.fld`, we first call `typeOf` on `expr`; this should return `DataType(x)`, where `x` is the definition of the struct. We then obtain `x` by calling a lattice function `getDataType` that unwraps the lattice value. Now that we

```
struct A{}
fn f(arg: A) {
  let tmp = A{};
}
```

Listing 16: Simple struct usage

have obtained the AST node of the struct, we find the field in the struct definition that matches the name of the field being accessed, and finally return the type of its type annotation.

```
protected def typeOf_fieldAccess(f : FieldAccess) : TypeLattice = {
  ty := typeOf(f.struct)
  s := TypeLattice.getDataType(ty)
  assert s instanceOf Struct
  fieldDecl := s.fields
  assert fieldDecl.name == f.name
  return typeOf(x.typeAnnotation)
}
```

Listing 17: Typing of field accesses

The type assignment for pattern match expressions is straightforward; it is simply the `lub` of all the right-hand sides of the match arms. However, the introduction of patterns somewhat complicated the typing of variables. We introduce a new function `expectedTypeOfPattern`, which relates patterns to their expected type based on their context. As can be seen in Listing 19, the expected type depends on what kind of AST node its parent is. The first three alternatives of the switch are straightforward: if the pattern is directly within a let-binding, it is typed in the way we discussed in the previous section. If it is inside a function argument, then it gets the type of that argument, and if it is inside a match arm, it gets the type of that pattern match expression. A pattern can also occur within a struct/enum pattern, as is the case for the pattern `y` in Listing 18. This is handled in the last body of the switch. We resolve the context to the declaration, and essentially perform a field lookup in that declaration.

Now that we have implemented `expectedTypeOfPattern`, the implementation of `typeOf` for variables is simple; for wildcards, including variable bindings, `typeOf` simply returns the result of `expectedTypeOfPattern`.

```
struct A{x: bool}
fn f(arg1: A) {
  match arg1 {
    A{x: y} => true,
  };
}
```

Listing 18: Simple pattern match

We extended `typeOf_withChecks` to check the typing rules related to pattern matches. The only noteworthy detail in its implementation is that it reuses `expectedTypeOfPattern`: it compares it output with the value returned by `typeOf`. It asserts that the types returned by these two pattern functions are compatible.

Conclusion In this section we introduced structs, enums and pattern matches. At this point our subset starts to resemble a usable language, and the fact that we were able to implement most typing rules is promising for the suitability of IncA for type checking. Unfortunately we also discovered

```
protected def expectedTypeOfPattern(n : Pattern) : TypeLattice = {
  parent := n.parent
  switch {
    assert parent instanceOf LetBinding
    return typeOfLetVar(parent)
  } alt {
    assert parent instanceOf FunctionArg
    return typeOf(parent.ty)
  } alt {
    assert parent instanceOf MatchArm
    grandParent := parent.parent
    assert grandParent instanceOf Match
    return typeOf(grandParent.expr)
  }  alt {
    assert parent instanceOf FieldPat
    grandParent := parent.parent        // get type of surrounding struct
    assert grandParent instanceOf PatternStruct

    variant := resolveRealPathToVariant(grandParent.path)

    assert variant instanceOf FieldsVariant
    field := variant.fields
    assert field.name == parent.name
    return typeOfFieldAccess(grandParent, field)
  }
}
```

Listing 19: Typing of patterns

that we can not implement exhaustiveness checking for pattern matches, but we delay its discussion to Section 7.1. All our subset of Rust misses right now to be actually somewhat usable is a lot of expression types, like binary operators. However, we are not going to implement those: they would be straightforward to implement and are therefore not interesting. Instead, we will continue in the next chapter with the more interesting features related to ownership.

Chapter 5

Ownership and borrow checking

This section discusses the implementation of Rust's most distinguishing feature: its ownership system. First, Section 5.1 introduces the affine type system, after which borrows are added to the language. The implementation of the typing rules for borrow checking is quite complex. As such, their discussion is split up into multiple sections. Section 5.2 discusses all rules relating to borrows except for the checking of lifetimes, which is split up into the next three sections. Lifetime checking requires keeping track of local borrows as they propagate through different variables. This is explained in Section 5.3, after which Section 5.4 discussed how explicitly named lifetimes are tracked. Finally, Section 5.5 builds on top of the previous two sections to implement the actual lifetime check.

5.1 Basic ownership

As mentioned in Chapter 2, one of Rust's goals is to provide memory safety. Key in this is to provide automatic memory management. It does so without the use of a garbage collector; instead, RAII is used to deallocate memory that can no longer be accessed. However, RAII by itself is not enough to prevent accidentally freeing the same heap memory twice (a so-called double free). The basic idea of RAII is to call an object's destructor once the variable goes out of scope. However, consider what would happen if in Listing 20 the assignment were to create a (shallow) copy of `x`. Then when the function exits, the destructor of both `x` and `y` would run, and any heap memory would be freed twice. To prevent such behavior, Rust combines RAII with the concept of ownership to guarantee memory safety.

```
fn f(y: X) {
  let x = y;
}
```

Listing 20: Moving a value

The idea behind ownership is that every value has a single variable that owns it, and a value can only be accessed through its owner. When a variable goes out of scope, all values that it owns are deallocated. This is done by a destructor which in most cases is automatically implemented. Since a value can only be accessed through the variable that owns it, once that owner goes out of scope, the values become unreachable. As such, its deallocation is guaranteed to be safe.

The mechanism described above relies on there always being a single owner for any value. This raises the question of what happens in an assignment like the one shown in Listing 20. At first glance it would appear that after the assignment the original value is available through both `x` and `y`, but this is not the case with Rust's semantics. There are two possible semantics of an assignment `x = y`. The assignment either *moves* or *copies* the value, depending on the type of `y`.

Moving values The first way an assignment `x = y` can behave is to *move* the value from `y` to `x`, together with ownership over the value. After the move, `y` can no longer access the value; attempting to do so would result in a type error. When in Chapter 2 it was mentioned that Rust uses a type system

based on affine types, it referred to this behavior of moving variables. An example of the semantics is shown in Listing 21. Note also that the read of the variable `x` causes `x` to become unavailable, even though it is not used in an assignment. Because of this, such reads are sometimes called *destructive*.

```
struct A{}
fn f(y: A) {
  let x = y;
  let z1 = y;   // error: read of moved value
  x;            // valid: a read of x, ignoring the value
  let z2 = x;   // error: read of a moved value
}
```

Listing 21: Variables with move semantics

When a new struct or enum is defined, it uses moving semantics by default. A destructor can be defined, which is automatically called when the value becomes unreachable. A value becomes unreachable when its owner goes out of scope, after its owner is destructively read, or its owner is overwritten by with another value. By being able to overwrite the destructor, Rust supports RAII: resources, such as heap memory, can be automatically released when it is longer reachable.

Copying values The second way an assignment `x = y` can behave is to *copy* the value of `y` into `x`. In this case, `y` maintains ownership over its variable, and the variable can be read any number of times. These semantics are demonstrated in Listing 22. The assignment `x = y` creates a new copy of the value of `y`, which can be observed through the assignment below it; after this assignment `y` will be `false` while `x` remains `true`. Values that have this copying behavior always remain on the stack, and no destructor is called when such a value becomes unreachable. Types that uses by-copy semantics include many primitives, including integers and booleans. Structs and enums can also opt-in to use these semantics by implementing the `Copy` trait (more on traits in Section 6.3), but this is only allowed if all of their fields also use copying semantics as well.

```
let mut y = true;
let x = y;        // makes a copy of the value of a
y = false;        // the value of b remains true
```

Listing 22: Copy semantics

Accessing uninitialized memory and checking immutability Rust's type checker statically prevents uninitialized variables from being accessed. Listing 21 already showed a case where a variable was illegally read twice. A slightly less obvious case is shown in `f1()` in Listing 23, where a variable is potentially read multiple times in a loop. If there were a break statement underneath the assignment, the code would be valid. This illustrates the need for a control flow sensitive analysis. Another case that can result in accessing an uninitialized variable is by declaring a variable without initializing it before use, as is shown in `f2()`.

The type checker must also ensure that immutable variables are assigned a value at most once. As such, it should throw an error on the second assignment in `g1()`, and likewise for the assignment in the loop in `g2()`.

Paths In addition to reads and writes to variables, accesses to fields within variables have to be considered. In Rust, a field can be accessed using the dot notation. For example, the expression `a.b` accesses the field `b` of variable `a`. Field accesses can be nested, e.g. `a.b.c`. We call a variable reference followed by 0 or more field accesses a *path*. Examples of paths are `a`, `a.b` and `a.b.c`.

When a path `a.b.c` is moved, as in Listing 24, it can not be accessed again; reading `a.b.c` again is illegal, and thus `a.b.c.d` becomes inaccessible as well. Furthermore, a value can only be moved

```
fn f1(a: A) {
  while some_condition {
    a;    // error: variable might be read multiple times here
  }
}
fn f2() {
  let a;
  if some_condition {
    a = true;
  }
  a;      // error: read of uninitialized variable
}
fn g1() {
  let a;
  a = true;  // valid
  a = false; // error; overwriting immutable variable
}
fn g2() {
  let a;
  while true {
    a = true;   // error
  }
}
```

Listing 23: Move and initialization checks

if the value is completely initialized. The paths `a` and `a.b` have both been partially moved and can therefore not be moved themselves. A path `a.b.e` could still be moved, since it is still accessible and fully initialized. If a move of a path x causes a subsequent move of a path y to become illegal, then we say those paths *conflict* with each other. It turns out the concept of conflicting paths is useful in many other analyses as well, as we will see later. The semantics used for reading a path depend only on the type of the innermost type. For example, a read `a.b.c` would use copy semantics if `a.b.c` is of a copy type, even though `a` or `a.b` might use moving semantics.

In the next subsections, we will discuss how the type checker implements checks for uninitialized variable access and writes to immutable variables.

```
struct A{b: B}
struct B{c: C, e: E}
struct C{d: D}
struct D{}
struct E{}

fn foo (a: A) {
  let x = a.b.c;
  // illegal to read: a, a.b, a.b.c, a.b.c.d;
  // legal to read: a.b.e;
}
```

Listing 24: Moving paths

5.1.1 Traditional implementation

The type checker in the official Rust compiler is split up into two phases. First, traditional type checking and assignment is done. If the first phase does not generate any errors, it continues with the so-called borrow checking phase, which does all the checks relating to ownership, and includes all control-flow sensitive checks.

The implementation of the analysis to check for uninitialized memory access is based on traditional set based uninitialized read analysis in data flow frameworks. In addition to *finding* errors, it also is able point to an expression that can cause the memory to be uninitialized. This can be the declaration of the variable, or a previous destructive read. Conceptually, a set of uninitialized variables is maintained during the traversal of the CFG, and whenever a path is either declared or destructively read, the path in question is added to that set, together with the location of the path (to be used for error messages). Whenever a path is written to, all instances of that path are removed from the set. When multiple paths join, the union of all sets is taken. An example is shown in Listing 25; the contents of the set of uninitialized variables *after* execution of each line is shown in the comments. The comments are mostly self explanatory. One interesting case is the loop on line 3*b*, which has two predecessors in the CFG. Initially, is assumes the result of 4*b* is the empty set. The CFG is traversed from line 3*b* to 4*b*, back to 3*b*, and finally to 4*b* for the second time, where it finds the first error. The algorithm can stop at this point, but for the sake of demonstration it continues. When the traversal continues to 3*b* for the third time a fixed point has been reached, and thus the algorithm proceeds to line 4*c*.

```
fn f(x: A, y: A) {
                  // 0   {}
  let mut z;      // 1   {(z, #1)}
  if true {       // 2   {(z, #1)}
    z = x;        // 3a  {(x, #3a)}
  } else {
    while true { // 3b  iteration 1: {(z, #1)}; iteration 2: {(z, #1), (y, #4b)}
      z = y;      // 4b  iteration 1: {(y, #4b)} iteration 2: error
    }
                  // 4c  {(z, #1), (y, #4b)}
  }
                  // 5  union of 3a and 4c: {(x, #3a), (y, #4b), (z, #1)}
  x;              // 6  (error)
}
```

Listing 25: CFG example

The checking of double assignments to immutable variables is done in a very similar way. The difference is that one item is added to the set whenever an assignment to a variable is encountered, and items are never removed from it. If an assignment to a variable is encountered that is already in the set (ignoring the line number), then an error is generated.

5.1.2 IncA implementation

Similar to the official compiler, virtually all analyses relating to ownership are separated from the typing analyses. This subsection discusses the analysis that checks for accesses of potentially uninitialized paths; the analysis that checks for invalid writes to immutable variables is very similar and is thus omitted. It turns out that IncA is very suitable for implementing the control-flow based set manipulation that is used in the traditional implementation; our implementation is very similar to the implementation in the official type checker, except that, for simplicity, we don't generate error messages. As such, instead of maintaining a set of tuples `(path, line)`, we maintain a set containing only paths. Also, we don't *explicitly* maintain a set. Instead, we make use of the fact that pattern functions in IncA can return multiple results; the returned items correspond with the items in the set of the traditional implementation. The analysis requires the traversal of the CFG, so we will first discuss how we implemented a pattern function to represent the CFG.

Control flow graph We represent the CFG by a pattern function `getSource`, which takes any CFG node, and returns all nodes that can precede the execution of the argument. To implement this, we use two auxiliary functions: `entryNode` and `exitNode`. Given a node `n`, these return the first and last nodes that are encountered in the execution of `n`, respectively. These functions are used extensively by `getSource`, since in general the control flows from the exit node of one node to the entry node of the next.

The function `getSource` is split up into different parts; one for each node type with non-standard flow control. Consider a method call `e1.f(e2, e3)`. Let `e4` represent the call itself. Then the order of the control flow is `e1` to `e2` to `e3` to `e0`. To be more precise, the control flows from the exit node(s) of one of these to the entry node of the next. The edge of `e3` to `e0` is straightforward to encode: we simply assert that the argument `n` is of type `MethodCall` and return the `exitNode` of the last argument if it exists; else we return the `exitNode` of the object (e.g. `e1`). The edge from `e1` to `e2` is implemented similarly, unlike the flow of `e2` to `e3`. The latter one is shown in Listing 26. Note that `n` is not being asserted to be of type `MethodCall`. Instead, the first line asserts that `src`, an unbound variable, is of type `ICFGNode`. Effectively, this causes the function to enumerate over all CFG nodes. By asserting that the parent of `src` is a `MethodCall`, we filter the enumeration down to only child nodes of method calls. Since we use `src.next`, we further narrow `src` down to items in a list. As the only list inside a method call is the list of arguments, we know `src` is an argument. We know that the control flows from one argument to the next, so the source of `src.next` is `src`. To be more precise, the source of `entryNode(src.next)` is `exitNode(src)`. As such, if `n` is equal to `entryNode(src.next)`, we return `exitNode(src)`, which is done in the last two statements.

```
private def getSource_MethodCall(n : ICFGNode) : ICFGNode = {
  assert src instanceOf ICFGNode
  parent := src.parent
  assert parent instanceOf MethodCall
  assert entryNode(src.next) == n
  return exitNode(src)
}
```

Listing 26: Partial implementation of getCFGSource

As mentioned, `entryNode` returns, for a given node `n`, the first node that is executed when `n` starts executing. Often, this is one of the children. For example, for a block it is the first statement in its statement list (if any), and for a function call it is the first argument (if any). Other nodes, such as a variable reference, return themselves. The function `exitNode` is similar, but returns the last node that has been executed after executing `n`. This is usually the node itself. For example, this is the case for variable references and function calls. The exceptions are blocks, for which the exit node of the last statement in its statement list is returned (if any), if-expressions which have exit points in both branches, and (while) loops, for which `continue` and `break` statements can be exit nodes. Blocks themselves are not included in the CFG unless it is an empty block. Now that we have discussed how we implement the control flow graph, we continue with the analysis for checking for uninitialized variable access.

Uninitialized variable access checking As mentioned, the idea for our implementation to check for uninitialized variable access is very similar to the traditional implementation. First we traverse the CFG to build the set of uninitialized variables for each point in the CFG. Then, we find accesses to variables, and if that variable is in the set of uninitialized variables, the surrounding function is marked as invalid.

Building the set of possibly uninitialized variables is done through the functions `varsBefore` and `varsAfter`. The function `varsBefore` returns all variables that might be uninitialized after executing any of its predecessors in the CFG. The result of this is used by `varsAfter`, which propagates each result of `varsBefore` that the current argument does not assign a new value to. This conditional propagation is corresponds to removing an item from the set in the traditional implementation. Additionally, `varsAfter` returns the variable x if the current argument moves x or declares x without initializing it. This corresponds to adding an item to the set. Simplified versions of these functions are shown in

Listing 27. Note that these pattern functions are mutually recursive. In general purpose languages, additional logic would likely be required to get the code to recurse until no new results are produced. In contrast, in IncA, this just works without additional effort of the programmer.

```
private def varsBefore(n : ICFGNode) : ICFGNode = {
  prev := getCFGSource(n)
  return varsAfter(prev)
}
private def varsAfter(n : ICFGNode) : ICFGNode = {
  // propagate previous results
  prev := varsBefore(n)
  assert undef initializes(n, prev)
  return prev
} alt {
  // return variables that are destructively read
  // or are declared without initializer
  return varsAt(n)
}
```

Listing 27: Finding possibly uninitialized variables

Finally, there is a pattern function that enumerates over all nodes `x` in a function, and if `x` is a read of a variable `y`, and `varsBefore(x)` returns `y`, then `x` accesses a potentially uninitialized variable and the function is marked as invalid.

This section introduced the moving and copying semantics, and we showed the implementation of the analysis that checks for uninitialized variable access. This analysis turned out to be quite concise and relatively straightforward; it seems IncA is well suited for such control-flow sensitive analyses that rely on set manipulations. The next section will discuss borrows, which make variable access more ergonomic.

5.2 Borrows

When a variable with move semantics is passed as an argument in a function call, the variable loses ownership over its value. If the caller wishes to regain ownership after the function call returns, the function could return the argument, as is illustrated in Listing 28. However, since it is so common to require only temporary access to values, Rust supports temporary references which it calls borrows. Borrows can be seen as an extension of the concept of ownership: they allow variables other than the value's owner to temporarily access a value without gaining actual ownership. There are two kinds of borrows: mutable borrows that have read-write access to the value, and immutable borrows that only have read access.

Borrows are conceptually very much similar to pointers in languages such as C. In fact, the syntax that Rust uses for them is very similar to C's syntax as well: a borrow of a variable `x` is created by `&x` for immutable borrows and `&mut x` for a mutable borrow. The same syntax is used in type annotations, e.g. a mutable borrow of a boolean has type annotation `&mut bool`. Like pointers in C, borrows can be dereferenced using the `*` operator to access the value that it borrows. A second way to create borrows is through the `ref` pattern Listing 29 shows these two ways of creating borrows. The statements in are `f` functionally identical to those in `g`.

Restrictions To be able to maintain the memory safety guarantee, various restrictions are placed on the usage and creation of these borrows:

1. A borrow can only be created of a value that is fully initialized.
2. A mutable borrow can only be created if the borrowed value itself is mutable.

```
fn f1(a: A) {
  // omitted: do something with a
}
fn f2(a: A) -> A {
  // omitted: do something with a
  a // return the argument
}
fn g(a1: A, a2: A) {
  f1(a1);
  // can not use a1 anymore

  // workaround to keep access to a2: return it as argument
  a2 = f2(a2);

}
```

Listing 28: Transferring ownership in function calls

```
fn f(mut arg1: bool, mut arg2: bool) {
  let x = &arg1;
  let y = &mut arg2;
}
fn g(mut arg1: bool, mut arg2: bool) {
  let ref x = arg1;     // equivalent to let x = &arg1;
  let ref mut y = arg2; // equivalent to let y = &mut arg2;
}
```

Listing 29: Two ways to create borrows

3. If a value is borrowed immutably, that value is *frozen* as long as the borrow remains in scope; that is, the value can not be moved or assigned to, neither by any of the borrows nor through the original owner.
4. If a value is borrowed mutably, access to that value is *claimed* by the borrow, which means that the value can only be accessed through that particular borrow until the borrow goes out of scope.
5. A mutable borrow can be used to mutate the borrowed data (e.g. `*x = true;`), but none of the borrowed data can be moved. This assures that borrowed content remains fully initialized at all times. Without this rule, after passing a mutable borrow as argument to a function call it would be unsafe to access any of the borrowed data, since it might have been moved.
6. There can be either a single mutable borrow or any number of immutable borrows of a value at a time. By enforcing this rule, data races are prevented since these can only occur when a variable is written to while it is accessed elsewhere. Multiple immutable borrows can be created by repeatedly borrowing a value, or, since immutable borrows act as copy-types, by simply copying existing borrows. On the other hand, mutable borrows have by-move semantics which helps to ensure that only one mutable borrow of a value can be active at a time.
7. The variable being borrowed must outlive the borrow itself. In terms of scopes, this means that the borrow must go out of scope *before* the borrowed variable goes out of scope. This restriction prevents dangling pointers. Discussion of this particular restriction is postponed to Section 5.3.

Borrows can be taken not only of variables but also of fields, in which case the practical effect of the restrictions become slightly more complicated. Consider a mutable borrow of a path `a.b.c`. Since Rust must be able to guarantee that `a.b.c` can *only* be mutated through the borrow, it forbids any mutations or borrows of paths that conflict (see Section 5.1) with `a.b.c`. This makes sense, since e.g. overwriting `a.b` would also overwrite `a.b.c`, and similarly, mutating `a.b.c.d` changes *part* of `a.b.c`. However, a mutation of `a.b.f` would not violate the restrictions. Various examples of what is legal and illegal when

a value has been immutably borrowed are shown in Listing 30, together with explanations of why they are or are not allowed.

```
struct A{b:B}
struct B{c:C}
struct C{d:bool}

fn f(mut a:A) {
  let b = &a.b; // create an immutable borrow
  {
    // these are all legal
    let tmp = &a.b.c;   // can immutably borrow nested fields
    let tmp = &a.b;     // can create another identical borrow
    let tmp = b;        // can copy borrows
    let tmp = &a;       // can immutably borrow the enclosing struct of `b`
    let tmp = a.b.c.d;  // non-destructive read through the owner
    let tmp = (*b).c.d; // non-destructive read through the borrow
  }
  {
    // illegal destructive reads
    let tmp = a;          // illegal; its field is borrowed
    let tmp = a.b;        // illegal; this exact path is borrowed
    let tmp = a.b.c;      // illegal; its enclosing struct is borrowed
    let tmp = *b;         // illegal; this is a destructive read of borrowed data
    let tmp = (*b).c;     // illegal; this is a destructive read of borrowed data
    // illegal conflicting mutable borrows
    let tmp = &mut a;     // illegal; its field is borrowed
    let tmp = &mut a.b;   // illegal; this exact path is already borrowed
    let tmp = &mut a.b.c; // illegal; its enclosing struct is borrowed
  }
}
```

Listing 30: various restrictions related to borrows

The type checker must know which borrows can potentially be alive at any point in the program; it needs this information to be able to judge whether or not certain expressions are valid. This is further discussed in Section 5.3.

5.2.1 Traditional implementation

Borrow checking in the official compiler is split up into three parts. The first check prevents mutable borrows of immutable data, i.e. restriction 2 discussed in the previous subsection. The rules for this check are fairly simple; the inference rules for this check are shown below. In these rules, `M(X, MQ)` indicates that `X` can be borrowed with mutability `MQ`, which is either `imm` or `mut` for immutable and mutable borrows, respectively. Rules 5.1 and 5.1 state that immutable variables can only be borrowed immutably, while mutable variables can be borrowed any which way. Rule 5.3 states that a path `a.b` is borrowable if and only if `a` is borrowable. Finally, rules 5.4 and 5.5 together disallow mutable borrows of dereferences of immutable borrows; i.e. it disallows `&mut *&x`, while it allows e.g. `&*&x`. These rules are implemented in the rust compiler using fairly straightforward recursion.

$$\frac{\texttt{Decl(X) = mut}}{\texttt{M(X, MQ)}} \tag{5.1}$$

$$\frac{\texttt{Decl(X) = imm}}{\texttt{M(X, imm)}} \tag{5.2}$$

$$\frac{\texttt{M(LV, MQ)}}{\texttt{M(LV.f, MQ)}} \tag{5.3}$$

$$\frac{\texttt{Type(LV) = \&mut Ty}}{\texttt{M(*LV, MQ)}} \quad (5.4) \qquad \frac{\texttt{Type(LV) = \& Ty}}{\texttt{M(*LV, imm)}} \quad (5.5)$$

The second check is aimed at guaranteeing the borrows can never outlive the values they borrow. This is further discussed in Section 5.3

The final check checks if the remaining restrictions can be uphold. For each borrow, the Rust compiler first calculates a set of restrictions, where each restriction is a tuple of an expression and a list of actions that must be prevented on that expression for the duration of the borrow. These restrictions are (at least conceptually) placed by a recursive procedure $Restrict(\alpha, \beta)$, where α is the expression being borrowed, and β is a list of actions that must be prevented on α. To reiterate the used terminology, a mutable borrow *claims* its borrowed value, while an immutable borrow merely *freezes* it. Since mutable borrows claim their borrowed data, it becomes illegal for others to read, move, mutate, claim or freeze the borrowed value, so it calls $Restrict(x, [read, move, mutate, claim, freeze])$. Likewise, an immutable borrow of x calls $Restrict(x, [move, mutate, claim])$, so x can still be read and immutably borrowed again (i.e. frozen). The *Restrict* procedure considers three cases:

1. $Restrict(\alpha, \beta)$ where α is a variable reference: simply record the restriction (α, β)
2. $Restrict(\alpha, \beta)$ where α is a field `x.y`: places a restriction (`x.y`, β) and propagates this restriction to `x` by calling $Restrict(\texttt{x}, \beta)$
3. $Restrict(\alpha, \beta)$ where α is a dereference of a *borrow*, e.g. `*x`: first a check is made to ensure that `x` outlives the newly created borrow. If `x` is a mutable borrow, a restriction (`*x`, β) is placed, and the restriction is propagated by calling $Restrict(\texttt{x}, \beta)$. On the other hand, if `x` is an immutable borrow, no mutation or claiming restrictions are propagated; these are already inherent in the type: the content of immutable borrows is never allowed to be mutated or claimed.

After the restrictions are determined, the type checker looks for expressions that violate an in-scope restriction. Expressions that can be invalidated because of these restrictions include variable reads, assignments and borrows. For expressions that use a path `a.b.c`, it also checks for borrows of prefixes of the path, i.e. `a.b` and `a`.

5.2.2 IncA implementation

Restriction 2 is implemented more or less the same as in the official compiler, so we will not discuss it further.

The remaining restrictions are implemented in a way that is somewhat different from the approach taken by the official compiler, since it is impractical in IncA to explicitly generate a set of restrictions for each borrow. Our implementation is split up into multiple parts, where the parts more closely resemble the restrictions we discussed as part the feature explanation. Many of these use the concept of conflicting paths introduced in Section 5.1.

One complicating factor is that dereferences and field lookups can be nested inside of each other. Our analysis simply ignores dereferences, i.e. `*((*((*a).b)).c)` would effectively be treated as if it were `a.b.c`. Unlike the official compiler, we effectively always propagate restrictions to the value that is being dereferenced. Since this is not always necessary, the official compiler does not always do this, but in these cases the restriction is ineffectual but not invalid. Furthermore, we were only able to find one example where an action is allowed on a dereference expression `*x` but not on the borrow `x` being dereferenced. This is shown in Listing 31; here, the official compiler allows the expression `&*arg` even though `arg` is mutably borrowed, while it disallows the expressions `&arg`. It turns out, however, that this is a confirmed bug in the official compiler. By ignoring dereferences, our implementation correctly disallows the `&*arg` expression.

The restrictions are implemented by several separate pattern functions:

- To prevent destructive reads of borrowed values, we enumerates over dereference AST nodes, and we determine whether the dereference causes a move. This is more complicated to get right than it would first appear. We have to take into account the type of the value being read, since copy types don't cause a move. Also, we have to check that the expression is not inside a borrow expression or is the left-hand side of an assignment expression. We even have to check that it is not bound to the `_` wildcard, since e.g. `let _ = x;` does not move `x`.

```
fn f(mut arg: & bool) {
  let a = & mut arg;
  let b = &*arg; // erroneously allowed in the official compiler
 // this would not be allowed: let c = arg; or let d = &arg;
}
```

Listing 31: Bug in the official compiler

- To prevent mutations of borrowed data that is not done though that borrow, we enumerate over expressions that mutate or move a path `x`; it then looks for borrows that are in scope and conflict with `x`. If any are found, the expression is invalid, since borrowed data would be mutated.
- The previous item already prevents the mutation of borrowed data. To ensure that mutable borrows have exclusive access to the value they borrow, the only additional restriction we need is one that prevent reads. As such, we enumerate over all expression that read a variable; if there is a mutable borrow of a conflicting path in scope, the read is illegal.
- To prevent mutations through immutable borrows, we enumerates over assignments. If the left hand side of the assignment is a dereference, and the value being dereferenced is an immutable borrow, the assignment is illegal.
- To assure that a value can have either one mutable borrow or any number of immutable borrows, we enumerate over pairs of borrows that are in scope at the same time. If the paths that they borrow conflict, and at least one of them is a mutable borrow, the borrows are illegal.

Many of these analyses need to determine which borrows are in scope at some point in the program. To determine this, it is necessary to keep track of which variables can contain which borrows. We will discuss this in the next section.

5.3 Data flow tracking

The borrow checking analysis described in Section 5.2 requires information about which borrows are alive at point in the code. This is not a easy as simply checking the scope a borrow is created in, since borrows can escape the scopes they are declared in. This is demonstrated in Listing 32; in both `f()` and `g()`, the borrow `&x` is reachable from outside the block they are created in. This means that borrows must tracked through assignments; some kind of data-flow analysis is required to keep track of which variables might contain which borrows. The data-flow analysis that Rust uses is somewhat overly conservative; consider Listing 33. If a control-flow sensitive analysis were used, the type checker could have figured out that the assignment `tmp1 = &y;` has no effect on the value of `p`; the scope of the borrow `y` would be limited to the inner block, and thus the borrow `&mut y` later on would be valid. However, this is not what happens in the actual compiler. Instead, the analysis appears to be control-flow insensitive. The reasoning for disallowing the `&mut y` borrow is that since `p` is assigned the value `tmp1`, and `tmp1` is assigned `&y`, then by transitivity `p` can contain `&y`. The `&mut y` borrow conflicts with the `&y` borrow stored inside `p`.

Borrows must also be tracked through dereferences, as is shown in Listing 34. Here `q` is a borrow of `p`, so the assignment `*q = &y` is effectively equivalent to `p = &y`.

Finally, borrows have to be tracked through fields. In Listing 35, the field field `a.c` has the value `&y`, so the assignments `let tmp = a.c; p = tmp` cause `p` to take the value `&y`, thus invalidating the `&mut y` borrow on the last line. If `tmp` were assigned `a` instead of `a.c`, the mutable borrow would remain illegal, since the `&y` would be reachable through `p.c`. On the other hand, if `tmp` is assigned `a.b`, the borrow `&y` is never assigned to `p` and is only reachable within the inner block. Therefore, in this case the mutable borrow `&mut y` would be valid. Field accesses are not always explicit. Besides directly binding variables, let-bindings can also use patterns. In particular, it can contain a struct pattern, thus deconstructing the initializer's value. An example of this is shown in the comments; here the pattern `A{b:_, c:tmp}` is used to deconstruct the value `a`. This declares a new variable `tmp` which is assigned the value `a.c`.

```
fn f(x: bool) {
  let y;
  {
    y = &x;
  }
}
fn g(x: bool) {
  let y = { &x };
}
```

Listing 32: Borrows that escape the block they are created in

```
fn foo(x: bool, mut y: bool) {
  let p;
  {
    let mut tmp1 = &x;
    p = tmp1;
    tmp1 = &y;
  };
  &mut y; // illegal even though p can not contain &y
}
```

Listing 33: Overly conservative analysis: the borrow &y is unnecessarily kept alive

5.3.1 Inca Implementation

The ultimate goal of the data flow tracking is to be able to query all borrows that are in scope. In order to implement this in IncA, we need a function that tracks value movement throughout the function. That is, we want a function that takes a variable and returns all values that it *might* contain. In a general purpose language this is fairly straightforward to implement, especially considering it is not dependent on the control flow graph. Such a function could simply record the values of all used paths. However, in IncA this is a bit more difficult. Consider the statement `A{b:_, c:tmp} = a`. Here, a general purpose language could simply record that `tmp` has the value `a.c`, whereas in IncA it is not as straightforward, since `a.c` does not appear directly in the AST. It might have been possible to implement this using lattices, but that would not have taken advantage of the incrementality that IncA provides. The actual function used in the IncA analysis is called `value()`, which returns all values that might be assigned to it at any point in the code. This function accepts not only paths as arguments but also dereference expressions such as `*x` and patterns, such as the left-hand side of the statement `let A{b:_, c:tmp} = a;`. Furthermore, it also operates on incomplete paths; that is, if the AST contains an expression `a.b.c`, results are also calculated for `a.b` and `a`. The function accepts such a wide range of arguments because it needs these as intermediate results.

The implementation of `value` is reflexive and transitive. In order to obtain the transitive closure, IncA's native support for transitive closures was used: we created a new function `transitiveValue`, which, given argument `n`, returns `value+(n)`. The function `value()` is recursive; at many places it calls `transitiveValue`. For brevity, in this section we refer only to the function `value()`, and assume it is a transitive closure.

The function `value(n)` is split up into several different pattern functions, each implementing the tracking of values through one kind of expression. Individually, each of these rules is not very complex. The difficulty in the implementation is to make sure that they keep working when different kinds of expressions are used throughout the program, even when they are nested inside each other. The implementations of the rules are described below. Afterwards, the section concludes with a discussion on how `value()` is used to obtain the in-scope borrows.

```
fn foo(x: bool, mut y: bool) {
  let mut p = &x;
  {
    let q = &mut p;
    *q = &y;
  }
  &mut y; // rightfully illegal: p will have value &y
}
```

Listing 34: Mutation through borrow

```
struct A<'a, 'b> { b : & 'a bool , c : & 'b bool }
fn foo(x: bool, mut y: bool) {
  let p;
  {
    let a = A { b: &x, c: &y };
    let tmp = a.c;  // or, equivalently: let A{b:_, c:tmp} = a;
    p = tmp;
  }
  & mut y;
}
```

Listing 35: Mutation through fields

Variable references The rule for variable references is straightforward: for each `a` and `b`, if `a` and `b` resolve to the same declaration, then they are equivalent to each other: `value(a) = b` and vise versa. Furthermore, if some variable reference `x` resolves to `y`, then `value(x) = y` and `value(y) = x`. Listing 36 shows how these last two cases are implemented: the first body simply resolves the variable reference and returns the declaration. The second body uses `tmp` to enumerate over variable references, and if that variable reference resolves to the argument of `value_varRefs`, that reference is returned. For brevity we omit the body that finds a variable reference that resolves to the same declaration as the argument of `value_varRefs`. We showed this example not because it is a particularly interesting implementation, but just so that the reader can get an idea of the structure of these pattern functions. As such, we will not show the implementations of the rest of the pattern values that together implement `value()`, and instead only give a higher-level description of the analyses.

```
private def value_varRefs(n : ICFGNode) : ICFGNode = {
  assert n instanceOf VarRef
  resolved := fieldAccess_resolveRoot(n)
  return resolved
} alt {
  assert tmp instanceOf VarRef
  resolved := fieldAccess_resolveRoot(tmp)
  assert resolved == n
  return tmp
} alt { /* omitted */ }
```

Listing 36: Partial implementation of value()

Patterns and path assignments The easiest case is the directly initialized let-binding, e.g. in the declaration `let x = y;`, the variable `x` clearly contains contains the variable `y`, thus `value(x)` will

return `y`. More generally, we say that outermost pattern contains the value on the right-hand side. For example, for `let A{b:_, c:tmp} = y;`, a call `value(A{b:_, c:tmp})` also returns `y`. Similarly, for the outermost patterns in pattern matches we return the expression that is being matched. This pattern function does not operate on values that are bound by reference. For example, in `let ref x = y;`, the value of `x` is not `y`; rather, the value of `*x` would be `y`. This is implemented in the next paragraph. Assignments to paths are implemented in the same way as let-bindings; for an assignment `a.b = y`, `value(a.b)` will return `y`.

Dereferencing The rules for patterns and assignments explained above are insufficient on their own. We describe the three additional rules for dealing with borrows below; each rule has a corresponding example in Listing 37:

1. If some node has a borrow `&x` as a value, then a dereference of such a node will have the value `x`. In other words, for any nodes `p` and `x`, if `value(p) = &x`, then `value(*p) = x`.
2. This rule is similar to the previous one, but applies when a borrow was created not with an explicit borrow expression `&x`, but is instead declared by a let-binding `let ref p = x;`. In such a case, a dereference of `p` has the value `x`, like before: `value(*p) = x`;
3. Variables can be mutated through borrows. The function `rule3` in Listing 37 shows a clear example of this: since `p` is a borrow of `x`, the assignment `*p = y` is effectively equivalent to `x = y`. This rule ensures that in this example, `value(x) = y`. More generally, for any nodes `p`, `x` and `y`, if `value(*p) = x`, and there is an assignment `*p = y;`, then `value(x) = y`.

```
fn rule1(x1: bool, x2: bool) {
  let p = &x;    // value(p) = &x
  let y = *p;    // rule 1: value(*p) = x
}
fn rule2(x1: bool, x2: bool) {
  let ref p = x;
  let y = *p;    // rule 2: value(*p) = x
}
fn rule3(mut x: bool, y: bool) {
  let p = &mut x;
                 // rule 1: value (*p) = x
  *p = y;        // rule 3: value(x) = y
}
```

Listing 37: Determining value of dereference expressions

Fields Tracking values through structs proved to be challenging. For an assignment `A{b:_, c:tmp} = a;` we would like to store the fact that the value `tmp` contains the value `a.c`. However, there is not a node `a.c` we can use for this. The same is true for assignments like `a = A{b:b, c:tmp}`, where we want to store the fact that `a.c` contains `tmp`. The key in our implementation is our recursive approach; the assignment may not contain a node `a.c`, but there may be one elsewhere in the code. For the `a` in that `a.c`, the result of `value` is `A{b:b, c:tmp}`. From there, it is easy to lookup the `c` field to get the value `tmp`.

Struct fields can appear in one of three ways in a function body: in a path expression such as `a.b.c`, in a struct initialization expression like `Foo{a: true}`, or in a pattern in a pattern match or let-binding, such as `let Foo{a: true} = (...)`. Path expressions can appear as both the left-hand side and right-hand side of assignments and let-bindings, whereas struct initialization expressions can only appear on the right-hand side, and struct deconstruction patterns can only appear as the left-hand side of let-bindings (and pattern matches). This makes for four possible combinations of field accesses, which is why the implementation of `value(e)` has four rules to track values through fields, which we enumerate below. For each of these rules, an example of the effect of the rule is shown in Listing 38.

1. **LHS = path, RHS = path**: if `value(x) = y`, then for any field access expression of field `a`, `value(x.a) = y.a`
2. **LHS = path, RHS = struct initializer**: if the value of `x` is some struct initialization expression `Foo{a:y}`, then `value(x.a) = y`
3. **LHS = struct pattern, RHS = path**: If a pattern deconstruction pattern `Foo{a:n}` has the value `x`, then for any field access expression of field `a`, `value(n) = x.a`
4. **LHS = struct pattern, RHS = struct initializer**: If the value of a pattern deconstruction pattern `Foo{x:n}` is some struct initializer `Foo{x:y}`, then `value(n) = y`.

```
 struct Foo{ /* omitted */ }
fn rule1(mut y: Foo, arg2: bool) {
  y.a = &arg2;
  let x = y;
  x.a; // rule 1: value(x) = y, so value(x.a) = y.a
  // Also, value(y.a) = &arg2, so by transitivity value(x.a) = &arg2
}
fn rule2(arg1: bool) {
  let x = Foo{a: &arg1};
  x.a; // rule 2: value(x) = Foo(a: &arg1), so value(x.a) = &arg1
}
fn rule3(y: Foo, arg2: bool) {
  y.a = &arg2;
  // rule 3: value(Foo{a: x}) = y, so value(x) = y.a
  let Foo{a: x} = y;
}
fn rule4(y: bool) {
  // rule 4: value(Foo{a: x}) = Foo{a: x}, so value(x) = &y
  let Foo{a: x} = Foo{a: &y};
}
```

Listing 38: Examples illustrating rules for tracking values through fields

Live borrows Now that we have discussed the components of the implementation of the function `value`, we continue with a discussion on how this function can be used to determine which *borrows* can be alive at any point in the code. First, the analysis determines which *variables* are alive. The analysis used for name resolution is reused for this purpose: a variable `x` is reachable from a node `y` if a variable reference at `y` resolves to `x`. Next, the `value` function is used to determine the values that these variables can contain. For variables that are of the borrow type, this suffices. However, variables can also be of a struct type, in which case an additional step is required to obtain the borrows nested within its fields.

If a variable `n` is of a struct type, and a borrow has been stored inside of it, then either the borrow has been assigned to `value(n)` through a field access expression, or `value(n)` has been initialized with a struct initialization expression. This motivates the implementation shown in Listing 39. First, notice that `nestedValue` returns the transitive closure of `_nestedValue`. The reason for this is simple: if `n` has a nested value `p`, then all nested values within `p` are also nested within `n`. The implementation of `_nestedValue` has three rules. The first is trivial; any value of `n` is also a nested value of `n`. The next body checks if the argument occurs within a field access expression; e.g. if the `n` occurs within a path `n.a`, where `a` is any field. Any values nested within `n.a` are also nested within `n`, so `n.a` is returned; the transitive closure ensures that all nested values within `n.a` are returned. Finally, if `n` is a struct initialization expression, then all fields within that expressions are returned. For example, for `Foo{a:x}`, the value `x` is returned. Again, the transitive closure ensures that any values within `x` will be returned as well.

```
private def nestedValue(n : ICFGNode) : ICFGNode = {
  return _nestedValue+(n)
}
private def _nestedValue(n : ICFGNode) : ICFGNode = {
  return value(n)
} alt {
  parent := n.parent
  assert parent instanceOf FieldAccess
  return parent
} alt {
  assert n instanceOf StructInit
  return n.fields.value
}
```

Listing 39: Analysis to get all nested values within the argument

Conclusion In this section we showed how we can track borrows that are created locally in a function. Compared to a traditional implementation, our IncA implementation is more complex. This is because our pattern function needs to use relations between AST nodes to track the data-flow, and there are not always AST nodes available that can directly represent a result. In Section 5.5 we will discuss how we use `nestedValue` to check for lifetime violations. However, in the next section we will first discuss how we can track lifetimes of borrows that are not locally created in a function body.

5.4 Named lifetimes

In rust, functions can accept and return values that contain borrows. When it does so, it must be parametric over the lifetimes used in the parameters and return value. Lifetime variables are declared within angled brackets, as is shown in Listing 40, where `f1` is parametric over the lifetimes `'a` and `'b`. The function accepts two borrows with lifetimes `'a` and `'b`, respectively, and return a borrow with lifetime `'a`. In this example it would be illegal to return `arg2`, since its lifetime `'b` might be shorter than the lifetime specified in the return type, i.e. `'a`. It is possible to specify lifetime bounds, as is demonstrated in `f2`. The syntax `'b:'a` is read "lifetime `'b` outlives lifetime `'a`", although strictly speaking it means that `'b` lives at least as long as `'a`. Since this is the accepted terminology, in the rest of this document the term *outlives* is used in this non-strict sense. In `f2`, it would be valid to return either `arg1` or `arg2`, since both borrows have a lifetime of at least `'a`. If a struct or enum uses borrows in its fields, these lifetimes must be explicitly named as well. The syntax to declare and use lifetimes is the same as for functions, as is demonstrated by `Foo`.

```
fn f1 <'a, 'b> (arg1: &'a bool, arg2: &'b bool) -> &'a bool {
  arg1
}
fn f2 <'a, 'b:'a> (arg1: &'a bool, arg2: &'b bool) -> &'a bool {
  arg2
}
struct Foo<'a> { d: &'a bool }
```

Listing 40: Declaration of explicit lifetimes

One of reasons that these named lifetimes are required is the fact that the caller of a function must know the lifetime of the returned value to be able to guarantee memory safety. For example, consider Listing 41. Here, the type checker knows that the return value of `f()` will be valid at least as long as the first argument, i.e. `&x` in the function call in `g`. As such, the scope of `result` is shorter than the scope of value it borrows, which means that the code is valid. In contrast, if the return type of `f` would

have been `&'b bool`, then the code below would have given an error, since `result` would outlive the value it borrows. This would be unsafe; variables are freed in the reverse order they are declared in, so `y` would be freed before `result`, temporarily leaving `result` pointing to deallocated memory.

```
fn f <'a, 'b> (arg1: &'a bool, arg2: &'b bool) -> &'a bool {
  arg1
}
fn g(x: bool) {
  let result;
  let y = true;
  result = f(&x, &y);
}
```

Listing 41: Usage of named lifetime

In addition to ensuring that the returned value has the correct lifetime, the type checker must also ensure that if any argument contains borrows, then the function maintains the lifetime restrictions specified in the type. For example, an argument of type `&'a bool` can not be overwritten with a borrow of type `&'b bool` unless `'b` outlives `'a`. Similarly, an argument of type `Foo<'a, 'b>` can not be overwritten by a value of type `Foo<'b, 'a>`

Sometimes, lifetimes in arguments and the return type do not explicitly need to be specified: there are some rules that determine whether or not these lifetimes can be elided. We successfully implemented these rules, but since both the exact rules and our implementation are quite tedious but not particularly interesting, we will not discuss these in this document. We continue this section with a discussion on how we can track lifetimes of arguments through different kinds of expressions.

5.4.1 IncA implementation

In Section 5.3 we discussed how values could be tracked through assignments. Using this, it is easy to query which borrows are alive, and what the lifetimes of these borrows are. However, this can not be used for keeping track of borrows that are within the function parameters. To see why this is the case, consider Listing 42. If `b` was a local declaration instead of a parameter, then `value(x.d)` would have returned the expression that created the borrow. However, since `b` is actually a parameter, there is no such initializing expression. Instead, we must somehow deduce that `x.d` has lifetime `'a`.

```
struct B<'c,'d> { c: C<'c,'d> }
struct C<'e,'f> { d: &'e bool, e: &'f bool }
fn f<'a,'b>(b: B<'a,'b>) -> &'a bool {
  let x = b.c;
  x.d
}
```

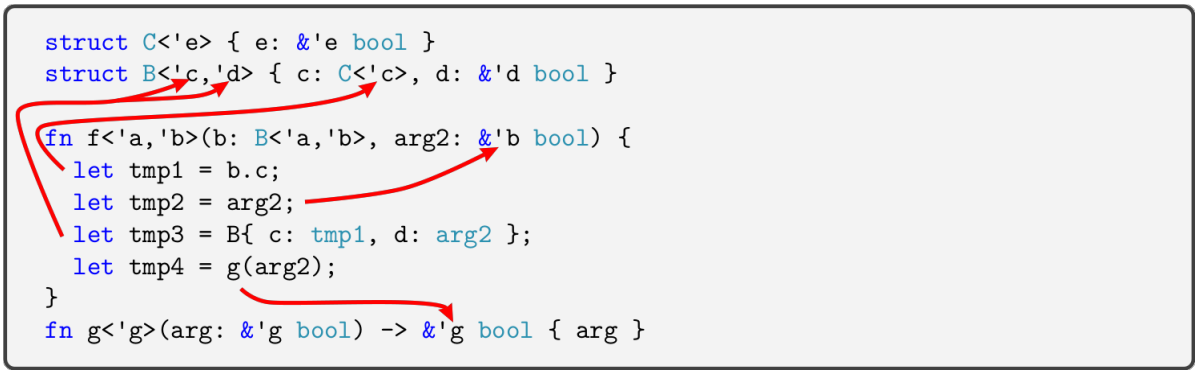
Listing 42: Tracking named lifetimes

What we need is a function that for a given expression returns a lifetime parameter that it uses, and the concrete lifetime it resolves to for that expression. For example, for the variable `x` in Listing 42, which is of type `C<'c, 'd>`, it should return the tuples `('c, 'a)` and `('d, 'b)`. For the expression `x.d`, which is of type `&'e bool`, it should return `('e, 'a)`. We will call this function `corresponds_expr`, which will return a tuple `(lifetime_id,resolved)`, where `lifetime_id` is a lifetime identifier, and `resolved` is the lifetime parameter of the function that it resolves to.

By necessity, the location of the lifetime identifier depends on the kind of expression. This is illustrated by Figure 1, which shows that field lookups must use the lifetimes declared inside the field, struct initializations use the lifetime parameters in the struct definition, references to arguments use the lifetimes inside the parameter type annotations, and results of function calls use the lifetimes inside

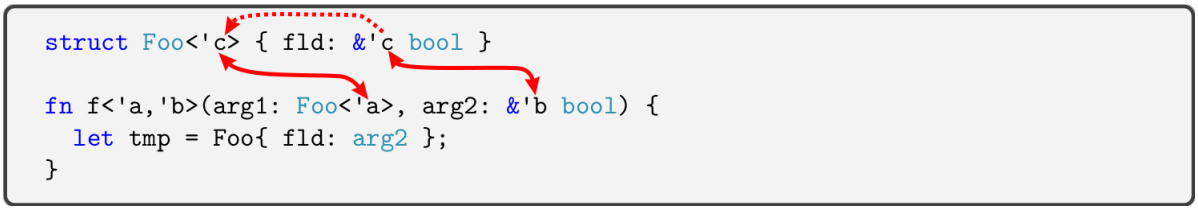
the return type of the called function. These are the only places which are guaranteed to be able to represent the types of these expressions.

Figure 1: The lifetime identifiers used by correspondsExpr



The implementation of `corresponds_expr` is complicated by the different lifetime identifier locations returned by recursive calls to `corresponds_expr`. For example, in Figure 2, `corresponds_expr(tmp)` must relate the lifetimes in its fields to its lifetime parameters (indicated by the dashed arrow). To do so, it must make a recursive call to `corresponds_expr(arg2)`, which will return a lifetime `'b`. Somehow, this `'b` inside the function must be related to the `'c` in the field of struct `B`. This link is implemented in the `lifetimeTwins` function. Given a type annotation or a struct definition, it returns tuples (a, b), where a is a lifetime inside the argument, and b is the corresponding lifetime in an equivalent type annotation or struct definition. For example, `lifetimeTwins(arg2)` returns `('b, 'c)`, indicated by the arrow on the right.

Figure 2: Identifying equivalent lifetimes



The implementation of `corresponds_expr` is split up into several different parts, each corresponding to an expression type. These are described in some more detail below.

Parameters and struct initialization The most basic case of `corresponds_expr(e)` is when `e` is a reference to a function parameter; for each lifetime `'a` in the parameter, it returns a tuple `(a, b)`, where `b` is the declaration of `'a`. In Figure 3, the returned tuple for `arg` is visualized by the dashed arrow. Struct initializations have already been briefly discussed, but to reiterate, when `corresponds_expr` is applied on a struct initialization, it relates lifetimes in the struct definition to its value. In the example, the result for `corresponds_expr(tmp)` is shown as a dotted arrow. To achieve this, for each field in the struct a recursive call to `corresponds_expr` is made find a tuple `(a,b)` for the field expression. In the example, `corresponds_expr(arg)` is called, which, as explained earlier, returns `('a, 'a)` as shown by the dashed line. The first lifetime in the tuple is related to a lifetime in the field definition using `lifetimeTwins`, as indicated by the solid arrow. This lifetime is then related back to the lifetime in the struct definition parameter, which is easy to do by simply comparing lifetime identifiers in the struct definition. It is indicated by the top-most arrow. Now the only thing left to do is to combine the results of the dashed arrow and the top-most arrow, resulting in the dotted arrow as intended.

Figure 3: correspondsExpr implementation for parameters and struct initializations

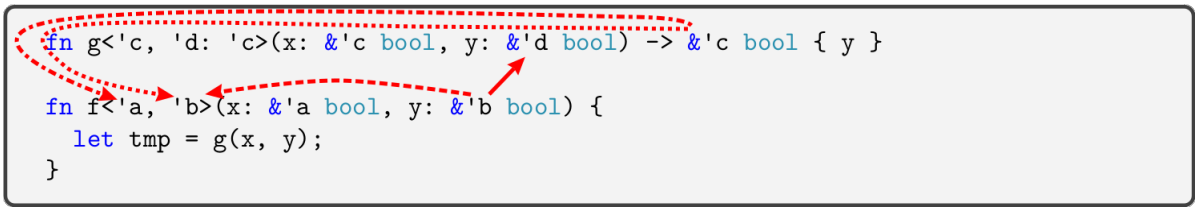
Field access Field accesses come in two forms: simple field lookups like `x.y`, and struct deconstruction patterns like `X{y:y} = x`. Both of these are processed identically. The idea is quite similar to struct initialization. Consider Figure 4, which demonstrates the steps taken to calculate `corresponds_expr(arg.b)`. The similarity to struct initializations can be observed by comparing this example with Figure 3; the difference is in the direction of the calculation, which now flows from the struct to its fields rather than the other way around.

Like for initializations, first a recursive call is made to `corresponds_expr(arg)`, which results in the tuple indicated by the dashed arrow. Then, `lifetimeTwins` is used to relate this lifetime to a lifetime in the struct definition (solid line). If that particular lifetime is used in the accessed field, as the top-most line indicates is the case, then the results of the dashed arrow and the top-most arrow are combined. This results in the dotted arrow, which is the relation that is finally returned by `corresponds_expr(arg.b)`.

Figure 4: correspondsExpr implementation for field accesses

Function calls For function calls, `corresponds_expr` will return a lifetime that is inside the function's declared return type. The procedure starts of similar to struct initializations and field accesses, but there is an interesting difference later on. The process is illustrated in Figure 5, where we one of the results of the `corresponds_expr` for the function call. Similar to the other cases, first a recursive call to `corresponds_expr` is made on passed arguments; we show the result for the `y` argument (dashed line). Afterward, `lifetimeTwins` is used to relate that lifetime to a lifetimes in the parameters of the called function(solid line). Finally, if there is a lifetime in the function's return type that outlives the lifetime in the parameter, then that lifetime is returned as part of the result of `corresponds_expr` (dotted arrow). We will not repeat the execution for the other argument, but its result is shown with the dashed-dotted arrow on the top. It might be surprising that call of `g` associates its return type with both `'a` and `'b`, but this is the correct behavior. This is because of the fact that in `g`, there is a lifetime constraint that states that `'b` outlives `'a`, while there is not in `f`. To be able to satisfy this constraint, the call in `f` must decrease the lifetime of the first argument. As such, the lifetime of the argument effectively becomes `min('a, 'b)`. Therefore, it makes that `corresponds_expr` returns both `'a` and `'b` for the call: both of these lifetimes must be long enough.

Figure 5: correspondsExpr implementation for function calls



Lifetime lattice Up unto this point, we assumed `corresponds_expr` returned a tuple of two lifetimes, whose second element represented the resolved lifetime in the function's lifetime parameters. It turns out this is not sufficiently expressive; there are two cases which can not be represented in this way. First, the lifetime may be a scope rather than an explicitly named lifetime. Such can be the case for function calls, as is shown in Listing 43. Here, the borrow `&mut tmp1` is invalid since `result` keeps the `&tmp` borrow alive. It is precisely this reasoning about borrows in the return values of function calls that requires binding lifetime identifiers to borrow scopes. For local borrows, the second item in the tuple that `corresponds_expr` returns is a borrow rather than a lifetime identifier: either an expression `&x` or a pattern `ref x`. It is important to note it is not possible to associate a borrow expression with a lifetime identifier right away, since there might not be a type annotation that represents the type of the expression. Tracking of borrows can only start at a later point in the code; the tracking starts at arguments to functions or struct initializations.

```
fn f1 <'a, 'b> (arg1: &'a bool, arg2: &'b bool) -> &'a bool {
  arg1
}
fn test() {
    let mut tmp1 = true;
    let mut tmp2 = true;
    let result ;
    result = f1(&tmp1, &tmp2);
    &mut tmp1;
}
```

Listing 43: Tracking local borrow through a function call

A second case is when the lifetime argument is implicit, as in Listing 44. Here, the lifetime used in the arguments for `f` are elided; that is, they are left implicit. To slightly simplify the rules for lifetime elision, when a lifetime is elided, it is assigned a unique lifetime. When the lifetime in a borrow type is elided, as is the case for `& bool` below, the editor assigns an invisible lifetime identifier to the type annotation. The point of having an AST node that is not visible to the user is to have an AST node that the analysis can point to. This way, the previous formulation of `corresponds_expr` keeps working. Unfortunately, this approach does not work for elided lifetimes in struct types, such as the elided lifetime of `x`. This is because the editor can not assign invisible lifetime nodes to struct references, since the amount of lifetimes in a struct depends on the implementation of the struct. Thus, an alternative approach is required. It is not sufficient to use a lifetime identifier of the referenced struct (e.g. `'a` in the code below), since multiple arguments of type `Foo` should each have their own unique lifetime. To assure that each elided lifetime is unique, a tuple is used which contains both the type annotation (e.g. `Foo` in the argument `x: Foo`) and the lifetime in the struct definition (e.g. the `'a` in `struct Foo<'a>`). Normally, when a function needs to be able return multiple different return types, you define an interface encapsulating all these types. However, in our case we need to use a lattice, since implicit lifetimes need to return two values instead of one. The lattice has three constructors; one for explicit lifetimes, one for implicit lifetimes and one for scopes of local borrows. The usage of a lattice is slightly overkill, since we don't actually do any aggregation on it. As such, its `lub` and `glb` functions are never called, and we can get away with throwing a runtime exception in their implementations.

```
struct Foo<'a> {a: &'a bool}
fn f(x: Foo, y: & bool) { // explicitly: fn g<'a,'b>(x: Foo<'a>, y: &'b bool)
  let y = x.a;
}
```

Listing 44: Implicit lifetime

5.5 Lifetime checking

This section discusses how data flow tracking, as introduced in Section 5.3 and named lifetime tracking as introduced in Section 5.4 are used to implement the actual lifetime checking.

A local borrow can only have insufficient lifetime when it is bound to a variable. As such, to find invalid lifetimes for local borrows we iterate over variables. We use the function `nestedValue` introduced in Section 5.3 to find borrows it can contain. To reiterate the rules for lifetimes of local borrows, if there is an assignment `x = &y`, then as long as `x` is in scope, `y` must be in scope as well. Otherwise, `y` would be deallocated, leaving `x` pointing to deallocated memory. In other words, `x` must go out of scope before `y`. Since variables go out of scope in the opposite order they are declared in, this is equivalent to requiring that `y` must be declared before `x`. This is exactly what our analysis checks for: if for any in-scope variable `x`, `nestedValue` returns a borrow `&y`, then if declaration of `x` is reachable from the declaration of `y`, we found an error.

Lifetime violations of named lifetimes are detected by a separate function. It is implemented in two steps. The first step generates constraints on expressions used in the function. This pattern function takes a function AST node, and returns a tuple containing three elements: an expression, a lifetime identifier, and the required lifetime. First, it places constraints on usages of arguments and on the return value, based on the lifetimes in their types. Then it recursively returns constraints on fields and dereferences of earlier constrained expressions. The second step takes a constrained expression and checks for violations of the constraint. The analysis function iterates over constrained expressions. A constraint can be violated in several ways. First, it is illegal for such an expression to be or contain a borrow of a local variable. Second, if the constraint requires the lifetime to be a specific implicit lifetime, then any borrows with different lifetimes are illegal. Finally, if the constraint requires an explicitly named lifetime, then any lifetime not outliving it is illegal. For example, in a function `fn f<'a, 'b:'a, 'c>(mut x: &'a bool, ...)`, the argument `x` is allowed to be overwritten by a borrow with lifetime `'a` or `'b` (which outlives `'a`), but not `'c`.

Conclusion In this chapter we introduced Rust's ownership. We expected the borrow checking to be challenging due to its control-sensitive nature. However, this was not the case: the control-sensitive parts of the analyses were not particularly difficult to implement. IncA is very suitable for such analyses. The data-flow analysis, which is not control-flow sensitive in Rust, turned out to be the challenging part. In case of the data-flow analysis of local borrows, the difficulty was caused by the fact that there are not always convenient AST nodes we could use to represent a result. Similarly, the tracking of explicitly named lifetimes is complicated by the fact that it is difficult to find AST nodes to represent the lifetimes in arbitrary. Ultimately, we were successful in our implementation.

Chapter 6

Polymorphic types

In this chapter we discuss the usage of polymorphic types. This is split up into three sections: Section 6.1 introduces generics, Section 6.2 discusses arrays and Section 6.3 discusses traits and the difficulties in implementing them.

6.1 Generics

Rust supports the usage of generic types in structs, enums, implementation blocks, and functions. In Section 5.4 we have seen functions and structs that are generic over lifetimes, and the same syntax is used for generalizing over types, as can be seen in the definitions of `Foo` and `g()` in Listing 45, as well as in the generic implementation block of `Foo`. The function `f()` demonstrates that expressions that use generic functions or structs require explicit values for the type arguments. In the complete version of Rust, these type annotations are optional when they can be inferred from the context. However, we were unable to implement the type inference required for this; we discuss this issue in Section 7.2. Fortunately, with the restriction that type arguments must be explicit, we *can* implement generics. We discuss this in the next subsection.

```
struct Foo<T> {x: T}

impl<T> Foo::<T> {
  fn h(self: & Self, arg: T) { }
}

fn f() {
  let x = Foo::<bool> { x: true };
  g::<bool>(true);
}
fn g<T>(arg: T) { }
```

Listing 45: Usage of generic types

6.1.1 Implementation

To be able to represent generic types in the type lattice, we extend the lattice with two new constructors. We will show the structure of these constructors based on the example shown in Listing 46, where we defined a generic type `Foo`. First off, the type that we assign to the struct declaration itself (i.e. `struct Foo`) does not change; it continues to be represented by `DataType(Foo)`. A specific instance of that struct, however, is assigned a different type. For example, the field `fld`, which has the type annotation `Foo<bool, bool>`, is assigned the new type `GenericInstance(DataType(Foo), [Boolean, Boolean])`. That is, we wrap the type of the struct with the constructor `GenericInstance`, which

additionally stores a list of type lattice values that represent the specific types of type arguments. Type arguments are not always concrete. For example, in the function `f`, which is generic in the type `V`, there is an argument with type annotation `Foo<bool, V>`. This `V` is represented in the type lattice as `Quantified(V)`, where `V` refers to the `V` in `f<V>`.

```
struct Foo<T, U> { /* omitted */ }
struct X{fld: Foo<bool, bool>}
fn f<V>(arg: Foo<bool, V>) { }
```

Listing 46: Example of a generic type reference

The construction of the type of an instance of a generic type is not straightforward, since this must be done in multiple steps. Consider again Listing 46; we want the node `Foo<bool, V>` to be assigned type `GenericInstance(DataType(Foo), [Boolean(), Quantified(V)])`. The basic idea is first to obtain the non-generic type `DataType(Foo)`, then to wrap it into `GenericInstance(DataType(Foo), [])`, and finally adding the types `Boolean()` and `Quantified(V)` one by one to the list of type arguments. We implement this in the recursive pattern function `genericItemType`, which is shown in Listing 47. To explain how this pattern function works, we will show how the type for `Foo<bool, V>` would be constructed. First off, `genericItemType` accepts two arguments. In our example, the first is the node `Foo<bool, V>`, while the second is a type argument; e.g. `bool` or `V`. This second argument is used only so we can recurse over the type arguments. As such, we assert that the type argument is actually one of the type arguments inside `Foo<bool, V>`; this is done by the assertion on the first line. While not strictly necessary, this line prevents the calculation of unused results.

The base case of the recursion is handled in the first alternative of the switch. Here, by asserting that `x.next` is undefined, we assert that `x` is the last item in the list of type arguments, as is the case for the value `V`. Now, we construct the type `DataType(Foo)` by resolving `Foo<bool, V>` to its declaration `struct Foo`, and calling `typeOf` on that declaration. Then, we call `typeOf` on `V` to obtain `Quantified(V)`. Finally, we call the lattice function `addGenericType`, which takes the `DataType(Foo)` and `Quantified(V)`, and returns the type `GenericInstance(DataType(Foo), [Quantified(V)])`. The recursive case of the recursion is handled by the second alternative. Here, the value of `x` is the `bool` in `Foo<bool, V>`. Since this is not the last item in the list of type arguments, we make a recursive call on the next item in the list, which returns the `GenericInstance(DataType(Foo), [Quantified(V)])` we constructed in the base case. Here, we use `typeOf` to obtain the type of `bool`, which will be `Boolean()`. Finally, we again call `addGenericType`, to combine `GenericInstance(DataType(Foo), [Quantified(V)])` and `Boolean()` into `GenericInstance(DataType(Foo), [Boolean(), Quantified(V)])`, which is the type we wanted to construct.

For convenience, we also implement a function `genericItemType_Wrapper` that only takes one argument; it calls the recursive function with its first argument. For example, `genericItemType_Wrapper(Foo<bool, V>)` calls `genericItemType(Foo<bool, V>, V)`. Finding the first type argument is done by calling `firstTypeArg` (not shown) since the first type argument is not necessarily the node that does not have a `prev` value; type arguments can be preceded by lifetime arguments.

A similar approach is used to type field accesses and function calls. Consider Listing 48. The procedure explained above is used to assign types to the `bar` field and the argument `arg`. However, to type `arg.bar`, a different approach is needed. Without generics, we could use the type of the field inside the struct definition. However, this would now return `GenericInstance(DataType(Bar), [Quantified(T)])`, and this would not be the type of `arg.bar`; since `arg` is of `Foo<bool>`, this `Quantified(T)` should be replaced by `Boolean`. More generally, we must replace all quantifiers by their actual type. To achieve this, we recurse over the type arguments in the struct declaration, e.g. over `A`, `B` and `C` in `struct X<A,B,C>{...}`. Each recursion replaces one of the type arguments by their concrete type. This is done by a lattice function that takes as arguments the type of the field `bar`, the type of the accessed variable `arg`, and finally the AST node representing the type argument (i.e. `T`) and its index (i.e. 0). In pseudocode, it is called like `Replace(Bar<T>, Foo<bool> T, 0)`. Here, it takes `Bar<T>` and replaces *all* occurrences of `T` by the 0th type argument of `Foo<bool>`, resulting in the correct type `Bar<bool>`.

```
private def genericItemType(t: TypeRef, x: ITypeRef) : TypeLattice = {
  assert x == t.pathArgs
  switch {
    assert undef x.next
    item := resolvePathToItem(t)
    ty := typeOf(item)
    x_ty := typeOf(x)
    return TypeLattice.addGenericType(ty, x_ty)
  } alt {
    ty := genericItemType(t, x.next)
    x_ty := typeOf(x)
    return TypeLattice.addGenericType(ty, x_ty)
  }
}
private def genericItemType_Wrapper(n : ITypable) : TypeLattice = {
  assert n instance TypeRef
  typeArg := firstTypeArg(p)
  return genericItemType(p, typeArg)
}
```

Listing 47: Typing of generic type references

```
struct Foo<T> { bar: Bar<T> }
struct Bar<U> { /* omitted */ }
fn f(arg: Foo<bool>) -> Bar<bool> {
  arg.bar
}
```

Listing 48: Typing concrete initializations of generic types

Conclusion Generics proved to be difficult to implement; we were unable to implement type inference for generic types, as we will discuss in detail more in Section 7.2. By requiring explicit type annotations when generics are used, we are able support generics. The types we assign to generics are the first types we can not construct in a single step, but we were able to build the required types in a recursive pattern function.

6.2 Arrays

In most languages the typing of arrays is very straightforward. In Rust, the goal of ensuring memory safety somewhat complicates the typing rules. The index used in an array element lookup can be an arbitrary expression, which means that the value of the index can not always be determined at compile-time. This makes it very difficult to ensure memory safety. The most obvious issue is that it can not be guaranteed at compile-time that the value of an array index is within valid bounds. Therefore, Rust must resort to run-time checking of these indices; if the index is out of bounds, an unrecoverable error called a panic is thrown which quits the program. This is, in fact, the only run-time overhead cost for maintain memory safety [3].

As discussed in Section 5.2, Rust's memory safety relies on the fact that the type system statically prevents conflicting borrows from occurring. However, the inability to determine the value of indices poses a problem: it is impossible to precisely track borrows *through* or *of* array elements.

Listing 49 shows an example where it is necessary to track borrows through array elements. Depending on the value returned by `getIndex`, `b` may or may not be a borrow of `a` and consequently `&mut a` may or may not be safe. Thus, to ensure memory safety Rust is forced to be conservative in the analysis of

borrow tracking: an element of an array is considered to possibly contain a certain value when that value is ever assigned to *any* element of the array. Similarly, Listing 50 shows an example where it is necessary to track borrows *of* array elements: whether or not the second borrow is safe depends on the values returned by `getIndex`. Again, Rust is forced to be conservative: an array element is considered to be possibly (mutably) borrowed if *any* element of the array could be (mutably) borrowed. Both of the shown examples must therefore be rejected by the type system.

```
fn f() {
  let mut a = A {};
  let mut b = & A {};
  {
    let mut c = [ &a, & A {} ];
    b = c[getIndex()];
  };
  & mut a;
}
```

Listing 49: a case where it is not possible to precisely track borrows through array elements

```
fn g() {
  let mut a = [ A {}, A {} ];
  let y1 = & mut a[getIndex()];
  let y2 = & mut a[getIndex()];
}
```

Listing 50: a case where it is not possible to precisely track borrows of array elements

The final problem related to indices is that it is impossible to keep track of which (parts of) array elements have been destructively read. This is solved by simply disallowing all destructive reads of array elements or fields nested therein.

In some cases the values of indices *can* be determined during compile-time. For example, if in Listing 49 `c[getIndex()]` were to be replaced by `c[1]`, then theoretically the program could be proven to be memory safe, but Rust makes no effort to catch such cases and would still produce a compilation error.

A final detail in the typing rules for arrays is that if the type of the elements stored in an array is a of a copy type, then the whole array is a copy type. Thus, the last letbinding in Listing 51 is valid despite the active borrows; it creates a new copy of the entire array. In contrast, if `arr` were to contain non-copy structs, the last let binding would be erroneous.

```
fn f() {
  let arr = [true, false];
  let borrowedElem = &arr[1];
  let borrowedArr = &arr;
  let arr2 = arr;
}
```

Listing 51: Example where an variable containing an array can be used on the right-hand side of an assignment even though it is borrowed

In Rust, the type of arrays not only includes the type of its members but also its size. The type of an array is denoted in Rust as `[T; N]`, where `T` is the type of the elements in the array, and `N` is the length of the array, which must be known during compile time. Two arrays with different sizes are

incompatible with each other. For example, an array of type `[bool; 1]` can not be overwritten by an array of type `[bool; 2]`.

Slices In addition to arrays, Rust also supports slices. With slices it is possible to refer to a contiguous sequence within an array without copying data. As can be seen in the example in Listing 52, the syntax to create a slice is very similar to normal indexing, but with a range rather than a single integer as index. Note that in the example, a borrow is used to create the slice. This is, in fact, mandatory for multiple reasons, the most important of which is that all local variables must have a statically known size. Again, since the exact range can not be determined at compile-time, the size can not be determined statically. While the expression `arr[1..3]` has an unknown size, the size of `&arr[1..3]` *is* known; it is simply the size of the borrow. In contrast to arrays which include the length in their type, slices do *not* include the length and thus have type `[T]`. Since the size of a slice can not be determined during compile time, it is a so-called Dynamically Sized Type (DST). While borrows of normal types are implemented as pointers, borrows of DSTs use so-called fat pointers; in addition to the normal pointer they contain the size of the type. In this case, the size of the type corresponds to the length of the sequence; this length is determined during run-time, and is used to implement out-of-bounds checks.

```
fn f() {
  let arr = [5, 6, 7, 8, 9];
  let slice = &arr[1..3]; // equivalent to &[6, 7, 8]
}
```

Listing 52: slice usage

The borrowing rules of slices are identical to the borrowing rules of normal indices.

While it is illegal to use DST's except through borrows, they *can* occur as field inside a struct (but not enum): the last (and only the last) field of a struct is allowed to be a DST (which implies a struct can contain at most one DST). A struct with a DST as a last field is a DST itself and thus must also be used through borrows. In that case, the size of the nested DST is determined through the size contained in the borrow (i.e. the fat pointer) of the containing struct. In practice structs rarely contain slices directly; code commonly exclusively uses slice *references* (borrows) rather than slices.

Creation There are two ways to initialize array in Rust. The first way is specify all initial items, e.g. `[true, false]`. The second way is a to use the repeat expression; i.e. `[e; n]`, where `e` is an expression and `n` is an expression that evaluates to an integer representing the length of the array. The result of this repeat expression is an array of size `n` where each element is initialized to `e`. If `n` is greater than 1 then `e` must evaluate to a value with by-copy semantics. As mentioned previously, type annotations have the form `[t; n]`, where `t` is the type of each element and `n` is the size of the array.

6.2.1 IncA implementation

In the TypeLattice, the type of arrays is `Array(TypeLattice, integer)`. Generating this lattice value for a repeat expression is straightforward, but the typing of manually initialized arrays requires a little bit of care. The index of the last element can be queried, but this value needs to be increased by one to get the array length. It is possible to define a function inside a Lattice to return an integer that, given some integer `i`, returns `i+1`. However, this would only work as expected when the model that is being analyzed happens to contain an AST node with the value `i+1`[1]. In general it is very dangerous to return arbitrary primitives from lattice functions; they should be wrapped inside some lattice. Instead of returning `i+1`, the lattice function should return a lattice value *containing* `i+1`, which is always safe to do. In this particular case, a function was made that accepts a TypeLattice `t` representing the element type and an integer `i` representing the last index, and that returns a lattice value `Array(t, i+1)`.

Another problem occurs when typing an empty array `[]`; while this expression is valid, it can not be typed without other context. In Listing 53, `array1` is assigned the type `[bool;0]` while `array2` is

[1]This restriction has recently been lifted from IncA

assigned type `[A;0]`. Typing this would require type inference, which as we will discuss in Section 7.2 we can not support. The workaround for this problem is the same as for generics: to require an explicit type annotation when initializing a zero-sized array. Fortunately, in practice zero-sized array are not very useful, so this is a reasonable concession to make.

```
struct A{}
let array1 = [];
let elem1:bool = array1[0]; // note: would give runtime error
let array2 = [];
let elem2:A = array2[0];   // note: would give runtime error
```

Listing 53: Type inference for arrays

The tracking of borrows of array elements is implemented by treating borrows of arrays elements as if they borrow the complete array. This works out very nicely: immutable borrows of elements can coexist with each other and with immutable borrows of the complete array, while a mutable borrow of either the complete array or one element of it disallows any other borrows to either the complete array or any elements thereof. Tracking of borrows *through* array elements is implemented by extending `value()` to treat all lookups on an array as equivalent to each other regardless of their index. Since the borrowing rules of slices are identical to the borrowing rules of normal indices, the same analysis code is being used to handle both cases.

The analysis to detect destructive reads on (fields nested inside) array elements is rather straightforward and thus will not be discussed here. While the analysis correctly disallows the *creation* of slices (when not wrapped in a reference), currently no restraints on the *usage* of DST types in structs, enum and parameters are implemented, since these are very uncommon in practice.

Conclusion The typing of arrays is somewhat unusual, since the type contains the size of the array. Fortunately, this turned out not to be very hard to implement. Typing of zero-sized arrays is more difficult; this requires type inference that we are unable to implement. Fortunately, zero sized arrays are rarely used. Although it seemed like a lot of borrow checking rules had to be implemented, this turned out to be easy to implement: borrows of array elements are handled as if they borrow the complete array. The next section will introduce trait, which prove to be a lot more difficult to implement.

6.3 Traits

One of the most important features of Rust is the usage of traits. At their core, traits are similar to interfaces in OOP languages such as Java: they are used to define shared behavior. Unfortunately, we were unable to implement a lot of features and restrictions related to traits due to time limitations. We will discuss this further in Section 7.3. In this section, we only discuss the features we were successfully able to implement.

Listing 54 shows an example where a trait `Foo` is declared and used. Functions declared inside a trait can optionally contain a default implementation, which objects can optionally override. The two structs `A` and `B` both implement `Foo`. The function `test1` demonstrates how trait functions can be called. In particular, we require all these calls to use the most explicit version of the function call syntax. However, this way of calling trait function offers no real benefits over regular methods. Traits are more useful when they are use in *trait objects*, as is demonstrated in `test2a`. This function accepts arguments of type `&Foo`, and we can supply any argument as long as it implements `Foo`. For example, `test2b` shows that it can be called with arguments of types `&A`, `&B` or `&Foo`. Also, the types of the arguments of `test2a` don't need two identical, as would have been the case if a generic function was used. When `a.f` is called, it call the function `f` in implementation block of either `A` or `B`. Which one of the two is called is determined during run-time; trait objects `&Foo` are implemented by using *fat pointers*. This pointer contains the locations of both the object's data and its virtual method table. Trait objects are technically dynamically sized, and thus must always be used through a borrow.

```
trait Foo {                        // declare a trait Foo
  fn f(&self, x: bool) -> bool;
  fn g(&self) { } // method with a default implementation
}
struct A{}
struct B{}
impl Foo for A {
  fn f(&self, x: bool) -> bool { false }
}
impl Foo for B {
  fn f(&self, x: bool) -> bool { true }
}
fn test1(a:A, b:B) {
  <A as Foo>::f(a, true);
  <B as Foo>::f(b, true);
}
fn test2a(a:&Foo, b:&Foo) {
  a.f(true);
  b.f(true);
}
fn test2b(a:&A, b:&B, c: &Foo) {
  test2b(a,b);
  test2b(a,c);
}
```

Listing 54: Basic trait usage

6.3.1 Implementation

As mentioned, we were unable to fully implement type checking for traits due to time limitations. Instead, we focused on trait objects and trait methods. While we did implement generic traits, we do not support generic *implementations*.

To be able to type trait objects, we extend the type lattice with the new constructor `Trait(x)`, where `x` is the AST node of the declaration of the trait. This type is assigned to both the trait itself and to any trait objects. For example, in Listing 55, both `trait Foo` and the `Foo` in the type annotation `&Foo` are assigned the type `Trait(x)`, where `x` is the `trait Foo` AST node. As such, this typing is very similar to the typing of structs and enums, and by itself is not that interesting; it is straightforward to verify that the call `f(x)` is valid, since the type of `x` exactly matches the expected type. However, the correctness of `f(a)` is more difficult to determine. While here the expected and actual types are not equal, this call *is* valid, since `A` implements the `Foo` trait. So far, our implementation checked the compatibility of the argument type with the parameter type within a lattice function, since this is the only way to implement structural recursion on the type lattice values. Such structural recursion is required, since trait objects can be located anywhere in the type, e.g. within a pointer `Pointer(false, Trait(Foo))`). There might also be multiple occurrences that need to be checked, e.g. in the type `Generic(DataType(B), [Trait(Foo), Trait(Foo))`. There is no easy way to use a pattern function to check all occurrences of datatypes and traits that are nested inside a type.

An ideal scenario, the lattice function could call a pattern function to get a set of all traits implemented by some type. Unfortunately, calling pattern functions from inside lattice code is not supported by IncA.

The type lattice function needs to be able to determine whether or not a type implements a certain trait. It would have been great if the lattice function could call a pattern function to determine this, but unfortunately this is not supported. As such, we to store a set of implemented traits inside the datatype. For example, `A` in Listing 55 would be assigned the type `DataType(A, {Foo})`, where the `Foo` indicates that `A` implements `Foo`. For non-generic traits this works great; we are able to correctly type check the example in Listing 55. To generate the set of implemented traits a power set lattice is used; this lattice wraps a set of trait declaration nodes, and the `lub` operation simply

```
struct A { }
trait Foo { }
impl Foo for A { }
fn f(x: &Foo) { }
fn g(x: &Foo, a: &A) {
  f(x); // straightforward to type check
  f(a); // valid, but requires us to determine whether A implements Foo
}
```

Listing 55: Trait object example

takes the union of both arguments. The aggregation is shown in Listing 56. While we omit the implementation of `getImplementedTraits`, this structure is still somewhat interesting; in particular, note that we unconditionally return `PowerSetLattice.bot()`, which returns the empty set. If the other alternative returns any result, this body has no effect. However, if no other results are returned, `allImplementedTraits` returns the empty set. This simplifies the calling logic; there is no need for additional logic to check for the case where no results are returned.

```
private def allImplementedTraits(x : DataType) : PowerSetLattice/lub = {
  trait := getImplementedTraits(x)
  return PowerSetLattice.singleton(trait)
} alt {
  return PowerSetLattice.bot()
}
```

Listing 56: Aggregating all implemented traits

Chapter 7

Problem cases

This section discusses cases which we were unable to implement due IncA's limitations. We start with the checking for non-exhaustive pattern matches. Then, we discuss type inference in the presence of generic types, and we finish this section with a discussion of the implementation of the type checking of generic trait objects.

7.1 Pattern matching: exhaustiveness checking

Rust supports pattern matches, an example of which is shown in Listing 57. Rust, being focused on safety, checks whether a match is exhaustive, and if it is not, it will refuse to compile. If any of the match arms in the example were left out, an error would be generated.

```
enum Y {
  Variant1 {
    item1: bool,
  } ,
  Variant2 {
    item1: bool,
  } ,
}
fn foo(y: Y) {
  match y {
    Y::Variant1{item1: true} => true,
    Y::Variant1{item1: false} => true,
    Y::Variant2{item1: x} => x,
  };
}
```

Listing 57: Pattern match example

7.1.1 Traditional implementation

Checking whether a pattern match is exhaustive can become a bit complicated when the matched value contains enums. It is not an option to simply generate all possible values and to check whether each is matched but at least one of the match arms, because there might be infinitely many combinations. This is the case, for example, when an enum of type `T` contains a reference to another value of type `T`. The algorithm has to be more sophisticated; the algorithm used by the official Rust compiler is based on Maranget's [18] work.

The input to the algorithm is a matrix that represents all match arms. In our subset of Rust, the initial matrix is a column vector where each cell is assigned one pattern. The algorithm is recursive,

and the input of the next recursion is calculated in one of two ways, depending on the first column of the input matrix: it either *specializes* the matrix, or it creates a new *default matrix*. These steps are explained below. The return value of the recursive function is a boolean that is true if and only if the match is inexhaustive.

Specialize If the first column of the matrix explicitly matches each possible variant without requiring (named or anonymous) wildcards, then the matrix is specialized. The specialization function takes the matrix `m` and a variant `v`, and computes the new specialized matrix `m2`. This new matrix is the result of transforming `m` in the following way:

- Rows in which the first element can not match `v` are removed
- Rows in which the first element explicitly matches `v` replace the first element in the row by its inner items, e.g. specializing `A{C{_}, E}` on `A` results in `C{_}, E`
- Rows in which the first element has the form `_` replace the first element in the row by a wildcard for each of the items inside the variant to which is specialized; e.g. specializing `_` for `A` results in `_, _`, since `A` has two fields

The match is inexhaustive if and only if for all possible variants `v`, specializing to `v` results in a matrix that is inexhaustive. That is, the disjunction of all possible specialization results is returned.

Default matrix If there exists a variant `v` that is not explicitly matched by any pattern in the first column (i.e. not counting wildcards), then a new default matrix is generated. This is created by removing all rows that do not start with a wildcard, and then removing the first column. The matrix is inexhaustive if and only if the new default matrix is inexhaustive. The reason why this works is that if a variant `v` is not explicitly matched by any row, then if the matrix is exhaustive, the rows starting with a wildcard must match any combination starting with `v`. But since these wildcards also match all other constructors, the explicit matches do not need to be examined.

At first glance, this step seems to be only an optimization; specializing to all possible variants would find the same solution. However, it also helps to prevent infinite recursion; e.g. when the enum of type `T` contains a value of type `&T`.

Base case If the input matrix has at least one row, but no columns, then it returns false. On the other hand, if the matrix contains no rows, then a pattern has been found that is not accepted by any of the match arms, and the function will return true (i.e. inexhaustive).

Example An example of an inexhaustive pattern match is shown in Listing 58; the pattern `A{C{G}, F}` is missing (note: we abbreviated the syntax for clarity):

```
enum Enum1{ A{ item1: Enum2, item2: Enum3 }, B}
enum Enum2{ C{item: Enum3}}
enum Enum3{ E, F, G}

match getEnum() {
  A{C{_}, E} => true,
  A{C{E}, F} => true,
  A{C{F}, F} => true,
  A{_,    G} => true,
  B          => true,
}
```

Listing 58: Inexhaustive pattern match

The input to the algorithm is the collection of patterns, stored in a matrix. For the example above, the matrix would be the following (column) vector:

```
A{C{_}, E}
A{C{E}, F}
A{C{F}, F}
A{_,    G}
B
```

The algorithm examines the first column, and since all variants of `Enum1` (i.e. `A` and `B`) are explicitly matched, the algorithm will perform specialization for both of these variants. First, it specializes to `A`, which results in the following matrix (which now has two columns):

```
C{_}, E
C{E}, F
C{F}, F
_,    G
```

Since `Enum2` contains only the `C` variant, which is indeed explicitly matched, the algorithm specializes to `C`:

```
_, E
E, F
F, F
_, G
```

The type of the pattern in the first column is `Enum3` which has 3 variants. Since `G` is not explicitly matched, a new default matrix is constructed:

```
E
G
```

Now `F` is not matched, and thus a new default matrix is created with size 0 x 0, and thus the algorithm will return true at this point. Since the specialization step returns the disjunction of all specializations, they can all stop directly and return true as well. Thus, the algorithm will have correctly identified that this pattern match is inexhaustive.

7.1.2 IncA implementation

We attempted to implent exhaustiveness checking, but were ultimately unsuccessful. The algorithm uses matrices, which can not be used in IncA. In an attempt to implement the algorithm in IncA anyway, it was attempted to write a function `inColumn`, which takes a pattern and returns all patterns that would be in the same column in one of the matrices. As the base case, for each top-level pattern, `inColumn` would return all other top-level patterns. Furthermore, for any pattern `p1`, if there is another pattern `p2` such that executing one step of the algorithm (i.e. a specialization step or creating a default matrix) on `p2` results in `p1`, then `inColumn(p1)` will additionally return the result of executing one step of the algorithm on any of the patterns in `inColumn(p2)`. In the end, this did not work out for various reasons, the most important of which is that a pattern can be part of multiple columns in different matrices. For example, see the matrix below. `A` would be in the same column as `B` if the first column was specialized to

`E`, while if the first column was specialized to `F`, it would be in the same column as `C`. Thus, `InColumn(A)` would return both `B` and `C`, which would lead to an invalid result.

```
_, A (specialized on D)
E, B (B would be in same column as A)
F, C (C would be in same column as A)
```

Another problem is dealing with the specialization of wildcards. Consider a `_` that matches a `Enum1`; when specializing for `A`, this would need to expand into `_, _`, but there are no AST nodes to point to in a pattern function. It is possible that this problem could have been avoided by, for example, making `specialize` on a wildcard return no result and instead letting `specialize` on a constructor return an additional result when the last field is matched (e.g. in the example below, letting specialize of `E` inside `C{E}` return not only `F`, but `G` as well).

```
C{E}, F
C{F}, F
_,    G
```

7.2 Polymorphic type inference

In Section 6.1, we added generics to our subset of Rust. However, we required that type arguments were explicitly specified in expressions that use generics. However, the complete version of Rust allows the omission of explicit type arguments when they can be inferred from the context. In other words, we allowed the usages in `f1` in Listing 59 but disallowed the usages in `f2`. This is because we were unable to implement the required type inference. In this section, we discuss the challenges of implementing type inference in combination with generics.

```
struct Foo<T> {x: T}
fn f1() {
  let x = Foo::<bool> { x: true };  // using explicit generic type
  g::<bool>(true);                  // using explicit generic type
}
fn f2() {
  let y = Foo { x: true };          // implicit generic type
  g(true);                          // implicit generic type
}
fn g<T>(arg: T) { }
```

Listing 59: Usage of generic types

The official Rust compiler uses Hindley-Milner-like [23] type inference, which makes use of a unification algorithm. The unification algorithm consists of two main parts: the first is to generate constraints on the types, while the second is to solve these constraints in order to obtain the final types. This second step involves, at least conceptually, calculating a the substitution table that associates type variables with actual types. It *might* be possible to generate the constraints in IncA using lattices. However, we see no way to efficiently incrementalize the process of solving these constraints.

While we can not efficiently implement a unification algorithm, there might be other ways to implement type inference. After all, before adding generics we did not need unification. However, adding generics to our subset of Rust significantly complicates the type inference process. Previously, a variable

`x` was typed by examining all assignments `x = y` including the initialization in the let-binding, and taking the `lub` of all types of `y`, as well as the explicit type annotation on `x` if there is one. It was not necessary to consider the *usage* of `x` to determine its type. If `x` had an explicit type annotation, or there was any assignment to it, then it was guaranteed that the complete type of `x` could be determined from the annotation and/or assignments. If there were no assignments to it, then its usage would be illegal anyway, since uninitialized variables are not allowed to be read, and thus its type would be irrelevant. With the introduction of generics, however, the complete type can not always be determined from an initializing expression. This is demonstrated in Listing 60, which uses the `Option` enum. The variable `y` is initialized to the value `None`, which only allows us to infer the type of `y` is `Option<U>` for some `U`, but the concrete type `U` remains unknown. Likewise, given that `unwrap` takes a `Option<T>` and returns a `T`, and that the type of `y` is `Option<U>`, we know that `x` must be of type `U`. Still, the exact value of `U` remains unknown. It is only through the usage of `x` inside a conditional that we can infer that `x` must be of type `bool`, which means that `y` must have type `Option<bool>`.

```
enum Option<T> { None, Some{val:T} }
impl<T> Option<T> {
  fn unwrap(self) -> T {/*omitted*/}
}
fn f() {
  let mut y = None;
  let x = y.unwrap();
  if x { }
}
```

Listing 60: Example where type inference is required

A second challenge in the implementation of type inference is that typing must be interleaved with the solving of the constraints. This can be seen in Listing 60; the variable `y` can not be assigned a type without resolving the method `y.unwrap()`, but the resolution of this method in turn depends on the type of `y`. Concretely, `typeOf(y)` should initially return `Option<U>`, where `U` indicates a type variable; this information is required to correctly resolve `y.unwrap()`, which is then used to refine the type of `y` to `Option<bool>`.

As mentioned above, with the addition of generics we need to consider the usage of a variable to determine its type. Consequently, in an assignment `x = y`, not only is the type of `x` dependent on the type of `y`, but the type of `y` is now also dependent on the type of `x`. Consider a program with the three assignments `a = y`, `x = y` and `x = b`. Before we added generics, we had already established that the inference of the type of `x` involved taking the least upper bounds of the types of `y` and `b`. Now, with the addition of generics, the type inference of `y` would involve taking the greatest lower bound of the types of `a` and `x`. As such, the analysis would contain inconsistent aggregation: it would nest a `glb` aggregation inside a `lub` aggregation (or vise versa). This is not allowed in IncA, since the algorithm that it uses to incrementally update aggregations requires recursive aggregations to be monotonic.

7.3 Generic traits and implementations

In Rust, Traits can be generic, as is shown in Listing 61 for the `Foo` trait. In Section 6.3 we changed the way we represent enums and structs in the type lattice: if a struct `X` implements the trait `Y`, then we gave `X` the type `Generic(DataType(X), {Trait(Y)})`. However, in the presence of generics it is no longer sufficient to store a list of implemented traits; the analysis must also know for which types the trait is implemented. For example, `A` implements `Foo<bool>` but not `Foo<char>`. As such, the call `f1(a)` is valid while `f2(a)` is not.

To be able to type check generic trait objects, we tried adding the type arguments for the trait to the typing lattice. That is, rather than giving `A` the type `DataType(A, {Foo})`, we tried assigning the type `DataType(A, {TraitImpl(Foo, [bool])}` to indicate that `A` implements `Foo<bool>`. At first, this approach seemed to work, but we ran into an issue when an infinitely large type would be constructed, causing the analysis to freeze. This is the case, for example, in Listing 62: `A` implements the trait `Foo<A>`;

```
struct A { }
trait Foo<T> { }
impl Foo<bool> for A { }
fn f1(x: &Foo<bool>) { }
fn f2(x: &Foo<char>) { }
fn g(a: &A) {
  f1(a); // valid;   A implements Foo<bool>
  f2(a); // invalid; A does not implement Foo<char>
}
```

Listing 61: Generic trait object example

the `A` inside this `Foo<A>` also implements `Foo<A>`, whose `A` implements `Foo<A>`, etcetera. As such, this approach does not work. One way to circumvent the issue is to not include the list of implemented traits for types that themselves occur within the list of implemented traits. That is, in `DataType(A1, {TraitImpl(Foo, [A2])}` the inner type `A2` will not contain the list of implemented traits. However, it is difficult to determine exactly when in the analysis the list of implemented traits should be included and when it should be omitted. Note that the construction of a self-referring type must be avoided at all times; it is not an option to first construct the complete type and then to remove the list of traits if it will not be used, as the analysis would freeze during the construction of the temporary type. Due to time limitation, this approach was not successfully implemented.

```
trait Foo<T> { }
struct A{}
impl Foo<A> for A { }
```

Listing 62: Problematic case in type assignment

There is an alternative approach to type checking generic traits that we have not tested yet. Instead of storing the list of implemented traits within the types themselves, an environment could be passed as a separate argument to the lattice function checking for type compatibility. For each data type, this environment would contain a list of all the traits it implements (and if the trait is generic, for which type arguments it is implemented).

Chapter 8

Benchmarking and validation

8.1 Testing framework and importing

To test the correctness of our analysis, we setup a test suite. This suite contains 780 tests. Each test is a single function, struct or enum. Items that must be checked are prefixed with either `test_ok` or `test_fail` to indicate that the type checker is expected to accept or reject the item, respectively. We wrote an MPS plug-in to verify the correctness of our tests; each item is exported to a text file, together with all other items in the surrounding module that are not explicitly expected to fail. This way, all of the item's dependencies are included in the export. The exported test is then checked by the type checker of the official compiler.

Because we implemented our subset of Rust in MPS, we can not directly edit Rust source files. To be able to import existing Rust code, we wrote an importer plug-in. The plug-in calls a small utility program that uses the *Syn* library to parse the Rust program and output the AST in a JSON format. The plug-in loads this AST using GSON. The structure of the AST produced by Syn does not exactly match the AST used by our implementation. This has two main reasons. First, since we support a smaller subset, some parts of the AST can be simplified. Second, in some cases a different AST structure simplified the specification of the editor behavior. To account for the differences in the AST structures, the importer has some additional logic to transform the Syn AST into our AST. While the importer was written to simplify the usage of real-world Rust code, this proved somewhat overly optimistic. Our limited subset of Rust does not support some features such as closures, and has only limited support for closures. Unfortunately, these features are heavily used in libraries, which means we were not able to import these libraries. Furthermore, since almost all real-world code relies on these libraries, we were only able to import a very limited set of programs. This is also the reason why we do not import the unit tests used by the official compiler.

If lattice operators are not implemented correctly, it is possible for an analysis result not to update correctly after a change in the AST. To verify that the incremental analysis updates in our analysis are correct, we ran the analysis, performed a number of random changes, and compared the result of the incrementally updated analysis with the result of a freshly run analysis of this changed code base. Since we can not directly compare the lattice values, our condition is that the exact same set of nodes must pass our type checker in both versions of the analysis. We ran this test with 10, 100, and 1000 sequential changes. The specific change we made are the same changes that we use for the benchmark; we will discuss these in the next section.

8.2 Benchmarks

To get an idea of the performance of the analyses we implemented, we setup a benchmark. In this benchmark, we run the analyses on our entire suite of tests, which, as mentioned, consists of 780 tests, which together make up more than 19000 AST nodes. We measure how long the initial analysis takes, and then make changes in the AST to measure the incremental updates. We implemented three kinds of changes:

- Swapping two expressions that are not nested inside each other.

- Changing the value of any string in the AST. Initially, a "_renamed" suffix is appended to the string. However, if the string already end with that suffix, it is removed instead. This way, we simulate both cases where name resolution changes from successful to unsuccessful and vise versa.
- Removing or adding a function. This action alternates between the two behaviors: the first time, it removes a random function. If this action is performed again, it instead adds the previously deleted function to a random location.

We believe that with these three different kinds of changes we exercise most parts of our analyses. We perform 1000 changes, where for each change one of the discussed actions is chosen at random. Specifically, we use a pseudorandom number generator with a fixed seed initialization. This ensures that the benchmark becomes reproducible. We use this fact to increase the accuracy of our measurements. We first run the benchmark twice to give the JVM time to warm-up. Then, we run the benchmark five more times, and for the initial analysis and each of the 1000 changes, we use the median of the time taken in those five benchmarks.

We are also interested in how the performance of our type checker compares to the official type checker. As such, we exported two version of our test suite to text format: one with all nodes, and one where tests that are intended to fail are removed. This is because the official type checker does not run the borrow checker if it has encountered a typing error in the type checking phase. Our test suite is split up into multiple modules (files), so we call the official type checker on each of them and measure the total time. The `rustc` compiler is called with the `-Zno-trans` flag, such that only the type checker (and borrow checker) is run. The `rustc` version we used is `1.24.0-nightly (8e7a609e6 2018-01-04)`, which was the latest version available at the start of the thesis project.

Results We ran our benchmark on a laptop running Linux with an i7-3630QM CPU, and measured the elapsed wall-time. The initial analysis took 47.395 seconds to run. On average, the analysis took 33.6 ms to update after a change. Of the 1000 changes, 47 took longer than than 100 ms to update, while the longest time to update was 434 ms. To give a better idea of the distribution of the update times, Figure 6 shows a Tukey boxplot of the update times, which means that the whiskers indicate the minimum and maximum values after outliers removed. To be clear, in a Tukey boxplot outliers are defined as those data points beyond the third quartile plus 1.5 times the interquartile range.

While the initial analysis took a relatively long time, most of the updates are very fast. It seems the analyses would be suitable for real-time feedback in an IDE. It should be noted this incremental performance was achieved without specifically optimizing any part of the analysis; there is likely room for improvement. A second point to note is that there is definitely room for improvement in the initial analysis. The output of IncA is a set of constraints, from which during run-time IncQuery constructs a computation graph. Until now, IncA analyses were relatively small, and the time taken to construct this graph was insignificant. However, with our more complex analysis, this construction actually takes up long time. We measured how long it took for an analysis to initialize on a program with only a single AST node, since in this case almost all time is spent on the graph construction. We measured it to be 28.084 seconds, which means that out of the 47.395 seconds that the initial analysis of our test suite took, less than 20 seconds are actually used for the calculation of the analysis results. Since there is no reason why construction of the computation graph should be done at run-time, an obvious improvement to the start up time could be achieved by calculating this graph during compilation.

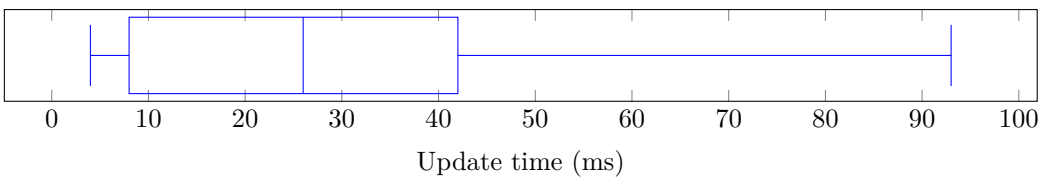


Figure 6: Tukey boxplot of incremental update times

The official type checker took 4.156 seconds to check the test suite including invalid functions, and 6.292 seconds to check the version without invalid functions. It should be mentioned that a comparison with our implementation is not completely fair, since we did not implement all features or even all typing rules for our subset of Rust. For example, our type inference is more limited. Having said that, the

difference in time between running the official type checker and running an incremental update of our implementation is significant. In fact, it is sufficiently large that at we can say with confidence that performance-wise, type checkers implemented in IncA are very promising for usage in IDEs.

Chapter 9

Related work

An official specification of Rust language or its typing rules does not exist. The typing rules of Rust used in this thesis were mostly inferred from the official Rust book [13] and the documentation and behavior of the official compiler. However, there *do* exist some formalizations of *parts* of Rust. Reed introduced Patina [24], a formalization of a subset of Rust that focuses on modeling Rust's ownership concept. It proofs that the formalized typing rules of this subset are sound and that they guarantee memory safety. Rust also allows unsafe code, which are pieces of code for which the restrictions on memory usage are lifted. The subset of Rust included in Patina does include the usage of such unsafe code. This was later addressed by Jung et al., who introduced a new formalization [12] of a subset of rust that *does* include the usage of unsafe code. They present a mechanically checked proof of memory safety in the presence of unsafe code. Both formalizations were not used in this thesis, since their typing rules are specifically constructed in a way that helps to proof safety aspects, and these formulations are not convenient to actually implement a type checker with. In fact, the second formalization does not even operate on the AST (or even the HIR).

There are several alternatives to IncA. One of the most obvious ones is Datalog [10], which originally was used as a database query language, but recently has been used in many other applications, including program analyses [11, 17, 25]. While these analyses are typically not used incrementally, there do exist algorithms to incrementalize the evaluation of datalog [7, 8]. There even exists a concrete implementation of an incremental datalog evaluator called LogicBlox [1]. Unfortunately, this seems to be the only concrete implementation and it has the drawbacks that it is a commercial program, and that its performance is not well documented [9].

Recently, Matsakis showed [20] that Rust's borrow checker can be implemented in Datalog. Matsakis' borrow checker certainly seems to be a lot more elegant than our implementation. It is able to be a lot more direct since it does not operate directly on the AST. Instead, it operates on the mid-level intermediate representation (MIR), which is created by transforming the HIR (which is close to the AST) to a small set of primitives. This reduced number of primitives simplifies the borrow checking rules. Furthermore, the MIR is created *after* type checking, and thus nodes in the MIR are already explicitly typed, conveniently providing lifetime identifiers for every expression. Also, Matsakis' implementation assumes certain inputs are available for usage in Datalog, such as a query that checks whether a given expression violates restrictions placed by a certain borrow. Finally, it does not attempt to exactly mimic the current behavior of the official compiler; its rules are slightly more permissive (without endangering memory safety).

Program analyses are somewhat related to database queries, as evidenced by the fact that Datalog has recently been applied to program analyses even though it was originally intended for database queries. Database queries are often implemented using dataflow operators such as map, filter and join. Research has been done to make queries incrementalizable in order to efficiently update the output of the query when the input changes. Furthermore, queries have been made more powerful by supporting looping constructs within queries. Of particular interest is recent work of McSherry et al. on differential dataflow [21], which provides both the incrementality and the expresivity provided by looping constructs. Here the incrementality is obtained by maintaining a set of differences in the input sequences of each iteration of the various dataflow operators. Recently, McSherry has shown in unpublished work [22] that Datalog queries can be implemented using differential dataflow, and that its incremental performance

seems quite promising. As such, it might be a viable alternative to IncA.

Another alternative to using IncA is to use attribute grammars [14]. Using attribute grammars, attributes are attached to AST nodes. The value of an attribute of a certain node is calculated during run-time, using attributes from its parent node (inherited attributes), and attributes defined in children nodes (synthesized attributes). Attribute grammars have in the past been used to implement type checking [5]. Later, Vogt et al. generalized attribute grammars to higher-order attribute grammars [29], where the value of an attribute can be an AST, whose nodes may contain also contain nested attributes. This can be used, for example, to define code transformations such as desugarings [4]. While (higher-order) attribute grammars can be evaluated incrementally [4], they have the downside that they can not be used when there are cyclic dependencies between attributes [29], which means they can not implement control-flow graph based analyses.

Typol [6] is a generalization of attribute grammars. In Typol, analyses such as type checkers are defined by inference rules, which nominally are compiled to Prolog predicates. However, these prolog rules can not be evaluated incrementally. Instead of compiling to Prolog, Attali et al. presented [2] an incremental evaluator that is able to incrementalize type checkers written in Typol. However, this evaluator is not able to evaluate inference rules that have a circular dependency.

Finally, an interesting approach to incrementalizing type checking is to use co-contextual type checking [9], where the typing rules avoid all context usages and instead propagate context constraints. However, co-contextual typing rules can become quite complex [15], and although it has been shown that the incremental performance gain can be significant, achieving that performance requires a lot of optimizations. Each new type checker that implements co-contextual type checking must manually implement such optimizations. Furthermore, some analyses, such as a method overload resolution fail to incrementalize efficiently using co-contexual type checking, whereas IncA is able to deliver a significant speed-up [27].

Chapter 10

Discussion

In this chapter we will discuss limitations of IncA and common patterns that we encountered. Finally, the last section concludes this thesis.

10.1 Limitations of IncA

The first, most obvious limitation of IncA is the fact that analyses can only enumerate over AST nodes (and some primitives). Sometimes, it would be useful to run an analysis for something that is not in the original AST. For example, lifetimes in function parameters can be left implicit. An example of this is shown in Listing 63. In the function `f`, the lifetime arguments of `Foo` are left implicit, which makes it a lot harder to write an analysis to track lifetimes, as we discussed in Section 5.4.1. It would have been very useful if IncA supported some form of desugaring or program transformation: if we could transform `f` to `g`, the analyses would be a lot simpler.

```
struct Foo<'a, 'b> { a: &'a bool, b: &'b bool }
fn f(x: Foo) { }
fn g<'a>(x: Foo<'a, 'a>) { }
```

Listing 63: Elided lifetimes

A second limitation that we encountered multiple times is the fact that negation in recursion is disallowed. We first discussed this limitation in Chapter 3. The intuitive reason for this limitation is that if a pattern function `g(...)` makes an assertion `assert undef f(...)`, then the results of `f` must be fully known before the result of `g` can be calculated. But if `f` in turn directly or indirectly calls `g`, a deadlock is created. This limitation corresponds to the concept of stratified negation in graph patterns [10]. We encountered this limitation multiple times. This was one of the reason for splitting the type assignment and rule checking into two separate functions `typeOf` and `typeOf_withChecks`. There are cases where `typeOf_withChecks` needs to call `typeOf` on subexpressions, and sometimes these functions are called with `assert undef`. This would result in negation in recursion if these functions were not split up.

Another limitation that we encountered is the monotonicity requirement on aggregations, which prevents combining `lub` and `glb` aggregations. As was discussed in Section 7.2, we encountered this issue when attempting to implement type inference in the presence of generics. Finally, pattern functions can not be called lattice functions, and as we mentioned in Section 7.2, the ability to do so could have helped a lot in the implementation of generic traits.

All in all, due to IncA's limitations, it is more difficult to write write analyses in IncA than it would be in a general purpose language. In fact, as discussed in Chapter 7, we were unable to implement several features. The most limiting factor is the fact that analyses must be written in terms of lattice values and AST relations. However, this limitation is fundamental for IncA to achieve its incrementality, and as such is unlikely to change. Some limitations do have workarounds; we will discuss common patterns in the next section, many of which are workaround for limitations of some form.

10.2 Common patterns in IncA

In this section we discuss a number of common patterns we encountered while implementing our type checker in IncA.

Universal quantification It is quite common to require that some pattern function is defined for all items in a list, or for all values returned by another pattern function. There are three ways to implement this, but each have their drawbacks. First, the analysis can be implemented recursively. An example of this is shown in Listing 64. Here, we want to assert that `isValid()` is defined for all arguments of a function call. The pattern function `funcCallOk` has two bodies: if there are no arguments, then the call is valid; otherwise, we find the first argument, and call the recursive function `argsOk` with the first argument. The function `argsOk` is defined if and only if `isValid` is defined for the argument and all subsequent items in the list. There are various disadvantages of this approach. First, it requires a lot of boilerplate code. Second, this approach only works on lists. That is, this approach can not be used to check that `isValid` is defined for all items returned by a pattern function.

Finally, the recursive nature has a negative impact on the incremental performance. For example, consider a case `argsOk` is defined for all arguments. Then, when an argument is appended to the end of the list, all the items in that list are reevaluated. This is because the `DRed` algorithm is used for updating results, which first propagates deletions and then rederives results. Since `assert undef arg.next` no longer holds for the now second to last item, that `arg` is deleted from the list of arguments for which `argsOk` is defined. Next, the argument preceding that one is removed since `assert def argsOk(arg.next)` no longer holds due to the deletion. This effect cascades all the way down to the first item. Then, `argsOk` is rederived for each of these arguments.

```
def funcCallOk(c: FunctionCall) {
  arg := c.args
  assert undef arg.prev
  assert def argsOk(arg)
} alt {
  assert undef c.args
}
def argsOk(arg: Argument) {
  assert def isValid(arg)
  switch {
    assert undef arg.next
  } alt {
    assert def argsOk(arg.next)
  }
}
```

Listing 64: Recursive implementation of universal quantification

The second way is to make use of De Morgan's law: instead of asserting that `isValid` must be defined for all items, we can assert that `isValid` not undefined any of the items. An example of this is shown in Listing 65. This has the downside that it uses `undef`, which in some situations can lead to negation in recursion. On the upside, there is less boilerplate required than for the other solutions. Compared to the previous solution, this one has better performance since the result for individual items do not depend on each other; e.g. adding a new item to the list does not invalidate the previous results.

A third way to implement universal quantification is the usage of lattice aggregation. We don't actually use this in our analysis, but it is still interesting, since it has benefits over both other options. Consider Listing 66. Here, `numValidArgs` returns `IntLattice.Value(1)` for each valid argument. It also returns `IntLattice.Value(0)` in case the function call does not have any valid arguments. These values are aggregated with the `sum` method, meaning the aggregation result is equal to the number of valid arguments. Then, `funcCallOk` compares that number with the number of arguments, which can be calculated from the index of the last argument. If the number of valid arguments is equal to the

```
def funcCallOk(c: FunctionCall) {
  assert undef argsNotOk(c)
}
def funcCallNotOk(c: FunctionCall) {
  arg := c.args
  assert undef isValid(arg)
}
```

Listing 65: Universal quantification using negation

number of arguments, they must all be valid, and thus `funcCallOk` is defined.

This way of implementing universal quantification can be seen as a compromise between the previous discussed methods. In contrast to the previous solution, this approach does not use `assert undef` on function calls, meaning that there is no risk of negation in recursion. Its is more efficient than the recursive solution, but less so than the negation-based solution. The aggregation results are maintained in a tree structure; if `numValidArgs` has `n` individual (i.e. non-aggregated) results, then a change in one of them requires updating O(log(n)) nodes in the tree. Updating a node consists of a single integer addition, so this is still quite efficient. It could be implemented even more efficiently by not using a tree at all; this is possible since the addition operator is invertible. A future version of IncA could add a syntax for universal quantification on lists, and it could automatically generate the aggregation.

```
def funcCallOk(c: FunctionCall) {
  assert undef c.args
} alt {
  assert IntLattice.Equal(numArgs(c), numValidArgs(c)) == true
}
def numValidArgs(c: FunctionCall) : IntLattice/sum {
  arg := c.args
  assert def isValid(arg)
  return IntLattice.Value(1)
} alt {
  return IntLattice.Value(0)
}
```

Listing 66: Universal quantification using lattice aggregation

Index An addition to the IncA language that was made some time after its initial release is the support for the index property, which is a property that all AST nodes that reside inside a list have. One of the most common uses of this property is for relating items of two lists with each other. For example, the function below relates the arguments of a function call to the function's parameters.

```
def f(c : FunctionCall) : (Argument, Parameter) = {
  func := resolveCall(c)
  arg := c.args
  param := func.params
  assert arg.index == param.index
  return (arg, param)
}
```

Listing 67: Common usage of the index property

Another, less common, use of the index property is for determining the length of list. For example, this is used to determine the size of a declared array (see Section 6.2).

Head Recursion In IncA it is possible to use recursion that would not terminate in normal imperative languages. For example, consider Listing 68, where `f` relates the arguments of a function call to the function's formal parameters. The first body simply relates the first argument of the call with the first parameter of the function. In the second body, recursion is used to obtain a previously generated tuple. Then, it returns a tuple consisting of the next items of both. The difference with Listing 67 is that depending on what `firstArgument` and `firstParameter` return, the indices of the items in the tuple might not be equal. For instance, in a method call, the self-argument is not in the argument list. As such, firstParameter might return the parameter with index 1 instead of 0.

In imperative languages, the second body would not terminate due to the lack of a base case stopping the recursion, but with IncA's semantics this is perfectly fine. In functional programming there is a well-known concept called tail recursion, which is characterized by the recursion happening at the end of the body. Since the type of recursion described above often happens at the beginning of the body, it could be called head recursion.

```
def f(c : Call) : (Argument, Parameter) = {
  func := resolveCall(c)
  return (firstArgument(c), firstParameter(func))
} alt {
  (prevArg, prevParam) := f(c)  // use head recursion
  return (prevArg.next, prevParam.next)
}
```

Listing 68: Example of head recursion

Interface casting Pattern functions can accept arguments that are of a interface type. We use this a lot in our analyses. These types can be narrowed down to specific types using an `assert x instanceOf Foo` statement. However, it is not possible to assert a node of a certain interface type to another interface type. For example, `f` in Listing 69 is illegal, since a node of interface type `IFoo` can not be refined to `IBar`, even though there may be nodes that implement both of these interfaces. As a way around this, we sometimes implement a helper function that accepts any type (which is called `BaseConcept` in MPS), refines that to the desired interface, and returns the result. An example of this is shown in `g` and its helper function `toIBar`. This may not be very efficient, but its usage can be convenient.

```
def f(x: IFoo) = {
  assert x instanceOf IBar
}
def g(x: IFoo) = {
  y := toIBar(x)
}
def toIBar(x: BaseConcept) = {
  assert x instanceOf IBar
  return x
}
```

Listing 69: Illegal type assertion

Pattern matching and casting Quite often, pattern functions accept an interface type as parameter, and each of the bodies starts with an assertion that the parameter is of a specific type.

Sometimes, there is a final body that acts as a default case; i.e. it asserts that it is not one of the previously asserted types. This is very reminiscent of pattern matching on the parameter, only a bit less convenient. Pattern matching is currently being added to IncA.

Another quite obvious pattern is that we often need to do an `instanceOf` assertion, only to use the variable only once after this casting. An example of this is shown in `getPatterns` in Listing 70. This is a very common pattern, yet the syntactic overhead is quite large. IncA is currently being extended to allow inline `instanceOf` assertions, as is shown in `getPatternsNew`.

```
private def getPatterns(e : ICFGNode) : Pattern = {
  parent := e.parent
  assert parent instanceOf LetBinding
  return parent.pat
} alt { /* omitted */ }
private def getPatternsNew(e : ICFGNode) : Pattern = {
  return e.parent:LetBinding.pat
} alt { /* omitted */ }
```

Listing 70: Syntactic overhead for instanceOf assertions

Conclusion IncA is a unique language, so it is no surprise that some unique patterns are encountered in its usage. For some patterns, IncA has been or will be updated to make the language more ergonomic to use. Many analyses were made more ergonomic by the introduction of the `index` property. For example, zipping two lists used to require head recursion, but by using indices, the code becomes more concise and easier to understand. Other patterns, such as universal quantification, point to potential areas of future improvements for IncA's ergonomics.

10.3 Conclusion

In this thesis we described how we implemented a type checker for a subset of Rust in IncA. We found that some analyses are a lot more difficult to implement in IncA than a a non-incremental version in a general purpose language would be. The most important reason for this is that pattern functions only operate on AST nodes, primitives, and lattices. It simply lacks the expressiveness of general purpose languages. However, it is exactly because of this limitation that IncA is able to achieve its incrementality. In a general purpose programming language, if one wants incrementality, it has to be implemented manually. In contrast, by using IncA, we hardly had to think about how to achieve good incrementality: as long as some care is taken in lattice function usage, the generated code has very good incremental performance. We showed in the benchmark that even in our non-optimized analyses, after a change in the AST, updating the analysis took only 34 ms on average, and more than 95% took less than 100 ms. In comparison, running the official type checker on a subset of the benchmark input that only contains well-typed items took 6.292 seconds.

We were not able to implement all of Rust features. This is partly due to time limitation, but there were also some features we were unable to implement due to IncA's limitations. These features include type inference in the presence of generics, the type checking of generic trait objects, and exhaustiveness checking in pattern matches. We discussed some of IncA's limitations and common patterns of IncA usage we encountered. In some cases, we suggested possible directions for future work on IncA. For example, a lot of the analyses would be easier to implement if IncA could support incremental program transformations.

In Chapter 1 we defined the two-fold goal for this thesis: first, to investigate whether IncA is suitable for implementing efficient incremental type checkers, and second, to serve as a case-study for Inca. We feel we accomplished this goal; the type checker we implemented does indeed have good incremental performance. Having said that, there were some features that we could not implement. As such, whether or not IncA is suitable for implementing a type checker depends on the target language. For simple languages, IncA is definitely suitable. For more advanced languages, such as those that support type

inference and generics, some features may not be possible to implement in IncA, so in these cases it is less suitable. If IncA can be extended to increase its expressivity, it would be very suitable for implementing efficient incremental type checkers in general.

Bibliography

[1] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382. ACM, 2015.

[2] Isabelle Attali, Jacques Chazarain, and Serge Gilette. Incremental evaluation of natural semantics specifications. In *International Symposium on Logical Foundations of Computer Science*, pages 87–104. Springer, 1992.

[3] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System programming in rust: Beyond safety. *ACM SIGOPS Operating Systems Review*, 51(1):94–99, 2017.

[4] Jeroen Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. PhD thesis, Utrecht University, 2015.

[5] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–116. ACM, 1981.

[6] Thierry Despeyroux. Executable specification of static semantics. In *Semantics of Data Types*, pages 215–233. Springer, 1984.

[7] Guozhu Dong and Jianwen Su. First-order incremental evaluation of datalog queries. In *Database Programming Languages (DBPL-4)*, pages 295–308. Springer, 1994.

[8] Guozhu Dong, Jianwen Su, and Rodney Topor. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):187–223, 1995.

[9] Sebastian Erdweg, Oliver Bračevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *ACM SIGPLAN Notices*, volume 50, pages 880–897. ACM, 2015.

[10] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. Datalog and recursive query processing. *Foundations and Trends® in Databases*, 5(2):105–195, 2013.

[11] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216. ACM, 2011.

[12] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Manuscript in preparation*, 2017.

[13] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[14] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[15] Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. A co-contextual type checker for featherweight java. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[16] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *International Conference on Compiler Construction*, pages 61–81. Springer, 2013.

[17] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

[18] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.

[19] Matsakis, Nicholas. Unification in chalk, part 1. http://smallcultfollowing.com/babysteps/blog/2017/03/25/unification-in-chalk-part-1/, 2017. [Online; accessed 21-August-2018].

[20] Matsakis, Nicholas. An alias-based formulation of the borrow checker. http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/, 2017. [Online; accessed 09-September-2018].

[21] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow.

[22] McSherry, Frank. Differential datalog. https://github.com/frankmcsherry/blog/blob/5201a5dd979632d2a4c8c07ee42fe65937e75129/posts/2016-06-21.md, 2016. [Online; accessed 15-September-2018].

[23] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[24] Eric Reed. Patina: A formalization of the rust programming language. *Tech. Rep. UW-CSE-15–03-02*, 2015.

[25] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog Reloaded*, pages 245–251. Springer, 2011.

[26] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: A dsl for the definition of incremental program analyses. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 320–331. IEEE, 2016.

[27] Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. Incremental overload resolution in object-oriented programming languages. 2018.

[28] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in datalog. *Proceedings of the ACM on Programming Languages*, 2 (OOPSLA), 208.

[29] Harald H Vogt, S Doaitse Swierstra, and Matthijs F Kuiper. *Higher order attribute grammars*, volume 24. ACM, 1989.