MASTER'S THESIS | LUND UNIVERSITY 2017

Purity checking for reference attribute grammars

Mikael Johnsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2017-22

Purity checking for reference attribute grammars

Mikael Johnsson (dat12mj2)
dat12mj@student.lth.se

July 5, 2017

LUND
UNIVERSITY

Master's thesis work carried out at Lund University .
Supervisors: Görel Hedin, gorel.hedin@cs.lth.se
Examiner: Boris Magnusson, boris.magnusson@cs.lth.se

Abstract

JastAdd is a meta compilation tool. It supports Reference Attribute Grammars (RAGs) and other declarative features for easy computation of abstract syntax tree properties in compilers. The attributes values are computed automatically on demand, and are defined by relations, called equations. JastAdd provides caching and optionally parallel evaluation for evaluation of attributes. The control flow is managed by JastAdd and thus not specified by the programmer. Therefore, the result must be invariant of exact order of execution. This requires that the equations are without side effects, i.e., pure otherwise unexpected behaviour might occur. This is not checked by JastAdd and upto the programmer to guarantee. For large programs keeping the code side effect free can be hard and side effect might find their way into the code and may cause subtle errors under specific situations.

I explored the possibility to use existing purity checkers for JastAdd. However, no existing tool was directly applicable, and I have therefore implemented a prototype purity checking tool for JastAdd and Java programs. The solution is then tested on several programs and the performance evaluated.

Keyword side effect, Reference attribute grammar, Abstract syntax tree, purity checker, declarative programming, JastAdd

Acknowledgements

I thank my supervisor Görel Hedin for suggesting the subject for the thesis and for commenting on the report. I thank Niklas Fors for providing an example program to test my tool on.

Contents

1 Introduction

Information on the purity of functions is useful for several applications. A method is considered pure, for this project, if it doesn't generate any externally visible side effects. This would correspond with allowing so called "benign" side effects [1] like internal caching of results and some changes to the own state opposed to more strict definitions.

Purity information can help with code optimizations and code management by giving a guarantee both to compilers and programmers that methods that are pure won't affect other parts of the program [1, 2]. This would allow them to be reordered more freely and make the program behaviour easier to analyse. Clausen showed how purity information can be used to perform compiler optimisations like dead code elimination, loop invariant code motion, constant propagation and common sub- expression elimination [2]. P. H. Nguyen also developed a side-effect analysis algorithm and tested it by performing compiler optimizations comparing with a different technique [3]. Purity can also help detect programming mistakes by failing to verify the absence of side effects when invoking a method intended to be side effect free. [1, 4].

The meta compilation system JastAdd [5, 6] uses a language that extends Java with declarative programming functionalities to enable the use of (Reference Attribute Grammars) RAGs. RAGs are an object-oriented extension of attribute grammar, a formalism described by D. Knuth[7] to calculate properties declaratively on abstract syntax trees. Knuth considered only the calculation of values but RAGs allows the use of references. References back into the abstract syntax tree (AST) allows the constructions of graph structures on top of the AST which enables different formulations of various types of analysis. G. Hedin has shown how RAGs be used for name and type analysis [8, 9].

JastAdd also supports several other declarative features beyond RAG with rewrites and circular attributes [9]. Circular attributes enable short formulations of several iterative problems including live analysis for variables. A full description of the features supported by JastAdd are listed in the documentation available on JastAdds homepage *www.jastAdd.org* [6].

Several aspects of JastAdd uses on-demand calculation and value caching assuming the calculations can be performed in any order with the same result. The execution order is not specified by the programmer but the order in which attributes are requested. To maintain the program behaviour well determined, side-effect free programming paradigm is desired for these aspects and a purity checker could assure this. Currently, JastAdd doesn't check if there are any side effects and simply trusts the programmer. There is a risk of introducing subtle errors that can be hard to detect.

The primary goal of this project is to attempt to use a purity checker to generate informative warnings about potential side effects in JastAdd generated code.

An JastAdd specification may contain normal Java code apart from the RAG. The side effect freeness requirement is only on the RAG components of the specification. This implies that three different sections of the JastAdd specification needs to be checked based on the assumptions made by JastAdd.

1. The specification of all attributes must be at least pure. Depending on the type of attribute it may also have to return a newly created object.

2. Java code invoked during attribute calculation must be pure. This is necessary for the attribute to pure.

3. Java code which uses an attribute may not "mutate" the "attribute value" but may do anything otherwise. A object obtained via an attribute may not be change but not necessary objects contained within such an object. The members of a lists may or may not be part of the "value" and the parameters.

The developers may then want to check code beyond the demands of JastAdd to guard an against mistakes or exclude some Java to allow some occasional exceptional behaviour. I decided to use annotations to specify the different restrictions to be imposed on different parts of the specification.

During this work, I check several previous purity checkers but they all have some obstacle to be readily applied for JastAdd checking. I then designed a new purity checker and annotation system to address these problems and tested it on Java and JastAdd programs.

The main purpose of this work is to introduce side effect analysis for JastAdd and generate warnings when there are side effects in code that are should have none. This includes the enumerated aspects and those the programmers specify.

2 Research Work

JastAdd generates Java code from the RAG specification. I decided to analyze the generated Java code instead of JastAdd source. This choice was motivated by two primary reasons. Firstly, since any purity checker for JastAdd must interpret Java since JastAdd uses both Java syntax and RAG syntax. Secondly a lot of previous work about side effect analyze has been done for Java as opposed to very little for RAG. It is then possible to compare the results with previous tools and research in the field. The research question, if any previous tool could easily be used to do the side effect analysis and generate the warnings, is only meaningful when there are previous tools.

To answer it the main challenges for performing purity analysis for JastAdd generated code must be explored. The addressed questions in this thesis could be summarized in the following points:

1. What aspects of JastAdd needs to be checked for side effects? How are these aspects translated into Java?

2. What challenges are there to perform side effect analysis for JastAdd?

3. Can a previously developed tool be directly used for the analysis? If not, how can a suitable tool be developed?

Most of this work considered the third point. I concluded that none of the tools are easily converted for the job and programmed a new solution for achieving side effect analysis for JastAdd. I test the previous tools in chapter 5 and chapters 6,7 presents an idea for an solution and describes my implementation. The tool is then applied to JastAdd constructions in chapter 8 which address part of the other question.

Contributions

This work provides several contributions to the field. The main points are listed below. I begin the report by presenting in section 3 some motivating examples and the sort of error message that could be provided. This is followed by some tool introductions and consents in section 4. From chapter 5 onwards the main work begins.

- In section 5 I test of previous side effect analysis tools. These including JPure [10], Efftp [10], Reim/Javari [11, 12] , IGJ [13] , ChAsE [14] , OpenJML [15] and Purano [16] to check if they work for the application. This consist of check capabilities such as which Java version they support and if they can handle some of the problem working with JastAdd entails. Providing an outline of obstacles for the appliance to the problem.
- In chapter 6 I present a solution for how to side effect check for JastAdd. For the solution I design a Java annotation system based on the annotations introduced in previous work combining aspects from JPure, OpenJML and Efftp along with extensions capable to expresses the required levels of purity for the problem.
- Modification of JastAdd to generate the annotations for RAG aspects so these can be purity checked. Annotations for the plain Java method are however not included and must still be in part annotated manually since only weakly inferred. I present these modifications in section 7.
- I Implement a basic purity checker for Java as an extension to ExtendJ [17, 18] providing features for the application. In section 7 I present this implementation.
- I compare the designed purity checker with the most closely related tool JPure and Efftp in sections 11 where I discuss related work.
- In section 9 I tested and attempted to test the purity checker on several compilers and Java programs. The tested programs include my own "SimpliC" compiler produced during the compiler course EDA65 [19], ExtendJ (previously JastAddJ) [17, 18], "forslambda", a small lambda calculus compiler by Niklas Fors who works on the Bloggi language and compiler also made with JastAdd [20] , JastAdd2 [5] and Fuji [21]. I also run the checker on itself.

In the end of the report in section 10 I shortly discuss my work and some problems or help I had. Followed by discussion of related work and suggestions for future works in sections 11 and 12. I finishes with a few concluding words about the results in section 13.

3 Motivating Examples

JastAdd doesn't check for side effects. This fact leaves open the possibility to make mistakes that might cause the code to work differently depending on the order attributes are evaluated in. This could result in unexpected bugs during some executions and triggered by changes in far separated parts of the program.

Example 1: A erroneous example of equations with side effects on a field. The order of counter() respective action() invocations effects the answer.

```
public static int ASTNode.counter=X;

syn int ASTNode.counter(){
        return counter++;
}

syn int ASTNode.action(){
        if (counter % 3)
                return 1;
        if (counter % 2)
                return 2;
        return 0;
}
```

In example 1 is a simple illustration of a set of equations where one of them has a side effect which effect the other. The attribute "counter" breaks the rules and has a side effect. Depending on if the attribute "counter" is evaluated before or after the attribute "action" a different result will be obtained from "action".

The example illustrates the possibility of using fields to influence the result of attribute. A possible scenario might be that field is given a correct value in one "initializing method" then used in other places. In this case attributes should not have been used since the assumption of evaluation order independent result doesn't hold.

It would be helpful if a warning message was generated for side effects like these stating where the problem is and what the problem is. For example a suitable error message would be as shown in example 2 which is the warning format I used for the purity checker I developed.

Example 2: A warning message for the side effect in example 1

```
In the synthesised attribute ASTNode.action()
declared in ...

lineNumber:counter++; contains side effects at counter.
Write to static field counter.

in AST/ASTNode.java:ASTNode
```

Another typical mistake is modification of a cached object. An attribute might calculate a mutable set which another attribute might then add a new element to. If the first attribute is later accessed after the modification the changed set is provided. The attribute no longer follows its specification. This scenario is exemplified in example 3.

Example 3: A erroneous example of equations with side effects. The second equation largerlist() modifies the first list obtained from list()

```
private List<Object> list;

syn List<Object> ASTNode.list(){
        return list;
}

syn List<Object> ASTNode.largerlist(){
        List<Object> list=list();
        list.add(this);
        return list;
}
```

Important special case for JastAdd is (Non Terminal Attributes) NTAs. A NTA is an attribute which adds a node or sub tree to the AST for which attributes can be calculated. The subtree must be fresh meaning newly created otherwise it will get the wrong parent. Attributes might depend on the nodes parent and consequently won't get correctly calculated. In example 3.

Example 4: NTAs must be Fresh. The example is only correct if r() creates a new Node and is used exclusively by the NTA equation

```
syn ASTNode ASTNode.r() = getChild(1);
// An ordinary reference attribute r that points to another
// node in the AST.
syn nta ASTNode ASTNode.f() = r();
// Wrong. The attribute f is an nta and its equation must
// compute a fresh ASTnode object.
```

4 Background

This chapter will describe some concepts used in this thesis along with the compilers JastAdd and ExtendJ.

I begin by presenting JastAdd which is the tool the purity checking should be combined with. After that I describe the ExtendJ Java compiler which I used as a basis for my purity checker. I then describe some theory about purity analysis, side effects, and purity checkers.

4.1 JastAdd

JastAdd is a meta compilation tool designed for constructing easily extendible compilers, analysers and similar tools.

JastAdd constructs classes for an (Abstract Syntax Tree) AST according to a specified grammar. An AST is a tree representation of a language. Different analysis is then enabled by inspecting the AST for specific properties. To simplify this process JastAdd supports inter-type declarations of both Java constructs and RAG attribute along with other constructs for performing calculation concerning the AST efficiently [22].

Inter-type declarations are properties like fields and methods inserted into Java classes when declared separately from the class declaration. This is done in a similar way to the approach in AspectJ [23]. The constructed Java class is the interweaving of all the declarations. The use of inter-type declarations allows for behaviours to be modularized into modules called "aspects". Using aspect oriented programming, new behaviours can be later simply added in new modules. An aspect is a container for a set of inter declarative declarations. Example 5 shows an example of an inter-type declaration where a field is inserted into a class from an aspect. The resulting class is then shown in example 6.

Example 5: A fields for the C class is declared separate in an aspect

```
public Class C{

}

aspect {
    public int C.x=10; // Interdeclarative declaration
}
```

Example 6: The resulting generated Java class

```
public Class C{
    public int C.x=10;
}
```

The RAG constructs are a type of inter-type declarations that defines *Attributes*. *Attributes* are like virtual methods. Their concrete implementations are called equations which must be side effect free methods[6].

JastAdd provides extra behaviours depending on the type of attribute and JastAdd settings which simplify working with the AST. The attribute can get extra code for caching result or traversing the AST to obtain or construct the value. JastAdd will convert RAG constructs to the necessary Java methods to calculate the value when the analyser is generated.

JastAdd constructs a tool by weaving together an abstract grammar description for an AST, along with normal Java source and then insert the Java code for all inter type declarations found in aspects. This generates a collection of Java classes representing the AST. A parser can be used to construct ASTs as objects of these classes.

All the RAG attributes should be purity checked to avoid any potential problems with unexpected behaviour when the JastAdd system calculate them. The control flow is not decided by the programmer so the code must support any control flow.

4.2 ExtendJ

ExtendJ is an extensible Java compiler implemented using JastAdd developed at Lund University under many years available at *http://www.extendj.org/* [18].

The use of JastAdd allows a new extension to be introduced using intertype declarations in new modules for the additional features. ExtendJ is built in layers of extensions successively adding support for new language features. Initially made for Java 1.4 but the first extended to Java5 by extending the features in java4 by G. Hedin and T. Ekman[24] and then later from Java 6 to Java 7 by J. Öquist and G. Hedin [17] . Erik Hogeman introduced basic Java 8 support in a new module for his 2014 thesis [25]. ExtendJ is also still being maintained[18].

Several extensions for ExtendJ are listed on ExtendJ home page[18] for example a checker for non-null expressions was made by T. Ekman and G. Hedinekman2007pluggable. Another extension called "multiplicity"[26] added some syntactic sugar for dealing with collections allowing among other things the add assignment operator (+=) for collection typed expressions. Finally, there 2 different extension providing control flow analysis to the compiler called Intraflow[27] and SimpleCFG.

4.3 Purity Analysis

Purity analysis solves the problem of discovering methods without any visible side effects or with limited side effects. Purity analysis/inference is also known as side effect analysis and the two terms are interchangeable [11].

According to David Pearce[10] and many other, side effect analysis is made most commonly using whole program interprocedural points-to analysis. This type of analysis is costly since they produce lots of data. A. Rountev [4] and A. Salcianu JPPA used variants of this approach along with many other.

Many different versions of points-to analysis have been applied working with different precisions and cost depending how accurately objects are tracked. There is those that are field based, variable based, type based in both flow insensitive and flow sensitive variants

to only mention a few. The flow sensitivity version considered the control flow within the methods while the flow insensitive doesn't.

The more sophisticated represent *access chain* more accuracy either "k-limiting" approaches which truncate after k-access levels or generalisations representing potentially infinite *access chains* like Geffkens *generalized access graphs* (GAG) [28] or A. Deutschs approach [29]. An *access chain* is a sequence of method calls or field accesses. For example "m().x.y.z" is an access chain of length 4. A 2-access level approach would truncate and merge that access with "m().x".

Two of the most well-known algorithms for points-to analysis are Andersen's algorithm [30] and Steensgaard's algorithms [31]. Andersen and Steensgaard calculate flow insensitive information with a cubic respective linear runtime [32]. Steensgaard's achieve almost linear time by being both field insensitive and having the aliasing relation be reflexive and transitive. This allows the creation of an alias-graph where variables are managed in group such that they only aliases one other group. The size of the graph is thus linear with number of variables. Andersen's however allow each variable to alias every other requiring a larger graph leading to the increased precision but longer runtime.

No matter which approach are use however it's not complete since it has been shown that points-to and alias analysis in general is undecidable according to Ramalingam and Landi [33]. This follows due to that the problem of deciding if any given path in a program is executable is undecidable. This mean that any approach necessary must make approximations and either fail to validate some valid programs or pass some invalid.

Method calls with dynamic dispatch can be modelled using class hierarchy analysis which tries to determine the type of each expression. Pearce used a variant called static class hierarchy analysis (SCH) which uses the static type of an expression to resolve which method can be invoked by a method call [34].

Pearce suggested the use of a modularly checkable type and effect system. Which would introduce that annotations summarise the conditions that must hold at function barriers, thereby removing the need for interprocedural analysis. By doing so the annotations determines how different variables may be used.

A type and effect system is any type system for reasoning about a program's computational effects. They have been applied for tracing a variety of different effects for different forms of analysis [35]. Examples including throwing exceptions and checking if eventually caught, data race analysis and memory manipulations and usage of IO and many more [35]. A type system works by introducing new annotations to the language and these annotations would describe effects and allowed behaviours.

Type and effect system such as Pearce's can limit the scope of inter procedural analyses by imposing conditions that hold on function boundary. Evaluation of each method in isolation is thus possible with only few facts having to cross boundaries. This allows for faster analyses inspecting only a part of a program instead of the whole program. Annotations can summarise the state of affairs when the associated language element is accessed or called.

The tools I checked have used several different forms of type and effect systems and many more has been researched. This includes reference immutability in the tool Javari [12] which has been combined with object immutability in IGJ(Immutability Generic

Java) [13] and a ownership system in OIGJ(Ownership Immutability Generic Java) [36].

Reference immutability is related to side effect analysis by preventing side effects by having immutable references. An immutable reference can't be used to induce any object modification side effects. A method's side effects can thus be limited by restricting the allowed changes via specific pointers. IGJ and OIGJ extends reference immutability with object immutability and ownership system. These extensions allow greater restrictions on aliasing and how values may be modified not to mention providing protecting for objects and not only pointers [36]. An immutable object is not modifiable from any pointer and the restriction provided by IGJ is deep and applies to all object reachable from an immutable object [13]. It is an indirect way of representing purity and the degrees of purity that can be represented are different from approaches more specialised for purity. REIM [11] is the only tool which explicitly formulated a purity definition used reference immutability.

Other tools use a memory access type systems that divides the heap into regions. These systems then constrain the side effects of methods to within regions or datagroups. An example is the datagroup based approach by Leino[37]. Plenty of research have been made but I find very few implementations for Java. Those I found focus mainly for checkers for Java Modelling Language (JML)[38].

Depending on the type and effect system used, the data flow across method and inside the method can be simplified.

4.4 Degrees of Purity

The authors of previously made tools have used several slightly different degrees of purity. They also disagree about what should be considered a side effect [1]. Different definitions exist about what constitutes a side effect. One definition is that a side effect is a change in some quantity not created by the current method.

Authors agree that modifying global field which is persistent memory are a side effect and direct assignment to local variables are not. Interpretation of parameter modifications are also a question that divides previous authors.

They also differ according to Stewarts article[1] in which effects they are interesting. Not every author considers throwing an exception or writing and reading IO as side effects. It depends on the type of application they had in mind for the system and the security concerns of the system. M. Finifter, A. Mettler used in Joe-E[39] that most effect are not allowed but an exception counts as a method result and may be passed along. A few exceptions are treated differently such as those related to running out of memory which may not be cached or thrown in Joe-E since they give too much state information. In another application throwing exceptions might be unacceptable for pure methods. Joe-E is a subset of Java to make reasoning about security properties of such systems or components easy.

In the articles [1, 40, 41]by R. Davies, X. Haiying, D. Naumann and others various definitions of side effects and purity are evaluated. The give different names for the levels of purity.

I have found 3 main different degrees of purity mentioned by several authors multiple times that I will reference in this work.

The first is *strong purity* that means the method doesn't perform any side effects at all. This means it doesn't assign any fields, create any object, call any native code, do IO operations or calls any method that isn't strongly pure. This corresponds with the strong purity used by Neumann [42, 41]. Haiying used a similar version of strong purity. *Functional purity* is a what M. Finifter, A. Mettler called it for their use with it in Joe-E [39]. The method may not read any field either.

Identification of strongly pure code is simple but too restrictive to be practically useful for most applications.

In many cases programmers are interested in weak purity and most tools can validate some aspects of the less restrictive *weak purity*.

Weak purity is achieved by allowing creation of new objects via weakly pure constructors, changing the newly created state of these objects and calling other weakly pure methods. This allow much more behaviours but requires more checking. Aliasing is now a problem since the checker needs to know that a given variable points to a newly created object.

Java Modelling Language (JML) is a language for declaring detailed "design by contract" (DBC) specifications where formal behaviour annotations declare the properties which should hold before and after a method's execution [38]. The DBC method allows programmers to reason about program behaviour and serves as a documentation about what exactly a method does given specific input relations.

Example 7: JML example with a pre- and post-condition. The precondition declares that the input parameter x must be less than 10 and the postcondition declares that the result is less than "10*m()."

```
/*@  public
@    requires x<10
@    \result <10*m()
@*/
public int JMLmethod(int x){
        return m()*x;
}
/*@Pure */
public int m(){
        return 1;
}
```

Pre-and post-conditions such as displayed in the example specify the method's behaviour and how it should be used. For such conditions to have well defined meaning and be verifiable, JML demands that methods used in the JML contracts must be pure. In the example, the method "m" must be pure.

The JML checkers (ESC/Java2, OpenJML)[38, 43] checks for at least partial weak purity with the *@Pure* notation. The Checker Framework [44] which is a framework for creating type checkers comes with a collection of type checker. Among these included

checkers there is a checker for an annotation call *@SideEffectFree* which is equivalent to JMLs *Pure*. Still the tests are so conservative that the verification is turned off by default.

Observable purity further relaxes purity by allowing a method to manipulate its own section of the hidden program state that is not visible from any allowed method call or field read by any client code on any visible object. It is this level of purity that is required for JastAdd since we want to support value caching and on-demand evaluation.

4.5 Purity checkers

Several different approaches to purity analysis for Java have been explored and have resulted in different tools. The tools I found and experimented with had different capabilities. The tool either contains a type checker, only does inferences or provides both type checking and type inference. For a few type systems I could find both a type checker and then a type inference tool.

Tools with only inference capability can only tell which methods are pure in a program. They don't check any annotations. These tools might be difficult to use for JastAdd due to the absence of any way to exclude side effects to cache fields. JastAdd also has side effects to fields dealing with attribute calculation such as keeping track of number of cycles modifying a circular attribute and if cycles occurs in the non-circular.

The Purity checkers works either with source code or bytecode. The purity checkers for bytecode would be able to work for both Java and Scala.

The type checkers which could be used for purity analysis are either designed to directly detect purity or which can obtain purity information by looking at some other properties such as reference immutability [12], object immutability [13] and ownership system [36]. There are also checkers that uses restriction of assignable locations.

The tested checkers can be collected into categories depending on the kind of type system the use. One category would be those that uses reference immutability, including ReIm, Javari, IGJ and OIGJ. All four was part of the Checker Framework and inspired each other with very similar annotations. Then there are those checkers that uses assignable heap regions which would include the OpenJML, ChAsE and in some sense JPure, Efftp and Purano. The last three uses a different but similar concept of modifiable localities. The tools JPure, Efftp, Purano and OpenJML provides additional check beyond modified locations.

Javari [12] is one of the reference immutability system and has an expressive representation for reference immutability differentiating also different levels in lists and arrays. The type checker however requires quite vast annotations of the source code and the Javarifier inference tool must be used to provide them. Javari was the original inspiration of all the other system I tested based on reference immutability.

The type checker ReIm[11] simplifies the type system from Javari. IGJ and OIGJ extends reference immutability with object immutability and an ownership system.

David. J Pearce made the tool JPure[10] that introduced and uses the concepts of freshness and locality of objects to analyse purity. Freshness is a concept introduced to denote that an object represent new state and assignment to its fields thus doesn't cause

any observable side effects. This was to avoid having to check the whole programs as a single unit.

The analysis in JPure is based on static class hierarchy analysis (CHA). Pearce's JPure inspired Efftp[45] for Scala that uses similar locality and freshness concepts but with generalizations and extensions. Efftp uses a flow insensitivity analysis for easier integration with several existing type checkers.

Alexandru D. S˘alcianu and Martin C. Rinards developed the tool "JPPA" [46] which uses a "pointer and escape analysis" that is supposed to also generate which field or function that caused a method to become impure. JPPA was made already in 2003. M.Geffkens developed a checker tool in 2014 which has similarities with JPPA depending on a similar but modified pointer analysis.

Purano by Jiachen Yang and others [16] is an interesting more recent tool which tries to balance both modularly checkable methods and the need for inter procedure checking. Purano preserves the ability to trace the side effects between methods.

This is just a small selection and there are several others that have described similar and alternative methods for example combining static with dynamic analysis. Several JML tools provide dynamic detection of purity by inserting runtime checks in the code.

I have however only considered tools which perform statical analysis since the goal is to warn during compile time and not during runtime testing.

5 Existing purity analysis

In this section I will test different purity checkers and attempt to use them for JastAdd. I will briefly explain the tools and some of the strengths of each tool but I focus mostly on the obstacles to their usage. I try to determine if the tools can be applied for the difficult situations imposed by JastAdd which I describe in more detail when I go over all JastAdd constructions in chapter 8.

5.1 Type checkers

The previous tools that could be used potentially by JastAdd applications are type checkers for effect systems [45] and reference immutability [11, 12, 47] possibly combined with object immutability [13] and ownership models [36]. JML tools such as Chase, ESC/Java2 and OpenJML support a memory access type system dividing the heap into datagroups and declaring them assignable or not.

Only Reim of the type systems Reim, Javari, IGJ and OIGJ have been directly applied for finding side effect free methods but all mentioned system can use the same or a more flexible version of method purity than the one used in Reim since Reim is the least expressive of these. These immutability type systems were compared by Johan Östlund and Alex Potanin in their 2013 paper [48].

Most of the used type systems have a way to declare that a method modifies only a specific part of the object's state. Javari, IGJ and OIGJ does that with their *@assignable* annotation and JML also has the *@modifies* annotation that can be checked with ChAsE[14]. JPure uses *@Local* for the same job. This functionality is useful for handling the caching in example 9 by having marked the only fields allowed to be modifications.

The idea when using any of these type checkers is to generate side effect warnings for JastAdd. The idea is that JastAdd generates the necessary annotations for the particular type system to type check if the generated code is sufficiently side effect free.

Example 8: JastAdd for the Cache Example. The compute functions should be pure

```
aspect Example{
   syn XType CacheExample.X() = new XType();
   syn XType CacheExample.Y() = X();
}
```

Example 9: Cache Example to annotate. The compute functions should be pure

```
public class CacheExample {
        XType X_value;
        boolean X_calculated = false;
        public XType X() {
                if (X_calculated)
                        return X_value;
                X_calculated = true;
                X_value = X_compute();
                return X_value;
        }
        public XType X_compute(){
                return new XType();
        }
        public XType Y_compute(){
                return X();
        }
}
```

To compare the different tools, I will annotate the small cache example shown in example 9 for each tool since handling caching in some way is necessary to handle JastAdd generated code.

This will show how the example can be annotated as pure and if the tool can handle the example at all. Further it is interesting to note how many annotations it takes. JastAdd contains many features that are more complicated to handle than this cache example such as managing circular calculation where a circular attribute must be allowed to call itself.

5.1.1 JPure

Pearce's JPure[10], made in 2011, uses the annotations *@Pure* , *@Fresh* and *@Local* for its purity type system. *@Pure* is placed on methods that are side effect free and *@Fresh* is placed on side effect free methods that return fresh objects. An object is fresh if it represents newly allocated state and is the state that an object is in after its constructor.

When creating a new object, a pure method is allowed to change certain fields in that object. These fields are said to be in the locality of the new object. These fields in turn refer to new objects, the locality of those objects transitively belongs to the locality of the original object.

The *@Local* annotation is placed both on fields that belong to an object's locality and on methods that only modify local fields. Pearce defines Locality rules that assure that if the parent object is fresh then so are all child objects in the locality. The calling context can thus modify the locality of objects that it has created without destroying purity [10]. In the article Pearce demonstrates how the locality concept allows JPure to handle the use of iterators and methods that store intermediate results in fields but it only gives

support for detection of weakly pure methods.

In the intended application for JastAdd-generated code the analysis must handle value caching but JPure is not up to the task. We would need to be able to annotate X() in the following example as *@Fresh* or *@Pure* but that won't type check in JPure since X() changes *X_value* and *X_calculated*.

Example 10: JPure cache example. The compute functions should be pure.

```
XType X_value;
boolean X_calculated = false;
@Pure public XType X(){
   if(X_calculated)
      return X_Value;
   X_calculated=true;
   X_Value = X_compute();
   return X_Value;
}
@Pure XType X_compute(){
...
}
@Pure XType Y_compute(){
   return X();
}
```

We could try with annotating *X_calculated* and *X_Value* as *@local*. However, then *X()* must also be *@Local* to be allowed to change them.

The protection against side effects is then unsatisfactory. For example, if *Y_compute()* modifies *X_value*, the type checker will not catch this error since we can't distinguish between the assignments to any *@local* field. Any calling context that calculates an attribute would be forced to either create a new ASTNode or to also be annotated with *@local*. The forced annotations are in the example 11.

Example 11: JPure cache example. Forced annotation due to type rules.

```
@Local XType X_value;
@Local boolean X_calculated = false;

@Local public XType X(){
   if(X_calculated)
      return X_Value;
   X_calculated=true;
   X_Value = X_compute();
   return X_Value;
}
@Pure XType X_compute(){
   ...
```

```
}
@Local XType Y_compute(){
   return X();
}
```

In order to handle the cache example satisfactory more annotations would be needed to be introduced. Introducing some heuristic detection of cache semantics along with a white list to solve the problem would probably be difficult and still other problems would remain.

The major drawback is the weaknesses of JPure's underlying compiler. JPure have the limitation of being old and build on an old version of a custom compiler also built by Pearce Jkit [10] that has only limited support for new Java constructs and doesn't support packages in given Java source but only in library class files. The authors of Reim found JPure fragile in their comparisons [11]. Error messages could however be generated from JPure by providing a list of methods that should be pure and then receive which methods that forces impurity of them.

Despite the limitations some form of comparison on examples that doesn't need to use caching or much benign side effects could still be made.

JPure also has the limitation of not giving any considering of arrays. Here there is need for an extension of the rules. When is an array fresh? Are the internal position part of the arrays locality or not? There is two ways that JPure could be extended to deal with arrays in order to work with variable length parameters. Alternative one is to have the element part of the locality and fresh as long on its known to only contain fresh objects. The other alternative is introduce extra annotations to include or excluded the inner objects.

The need to address this problem is because arrays are commonly enough used to preform simple changes to many objects. The question about arrays freshness is particularly problematic in relation with variable length arguments which is currently not addressed by JPure. Therefore, JPure would need a significant overhaul in order be usable. A similar limitation exists with library methods for which it is not possible to annotate fields in library classes. This leaves the programmer with no way to specify if internal objects in a return collection may be changed or not.

In general, I think JPure's annotations system is quite decent for the application but some extensions are needed to deal with more cases and for Java constructions which has been introduced since the tool was made. I think it's better to reimplement than try to modify JPure due to the limited underlaying compiler.

5.1.2 Efftp

Efftp [45] is a tool inspired by JPure for analysing Scala source. It generalizes and tweaks the type system of JPure in several ways. In Efftp every method can have a parametrized locality annotation for the returned object in which the locality invariants hold. The locality annotation denotes the locality the method returns and not the localities modified as JPure does. Instead there is a new annotation for modifications *@mod* which denotes

the localities that may be changed inside the immediate method body and thus fills the role of JPure's *@Local*. A method annotated *@mod(this)* would be allowed to do the equivalent to JPure's *@Local* method. Efftp allow modified locations to be specified more accurately than JPure with an annotation like *@mod(this.x,a,b)* giving the exact location.

Efftp can return all the localities the method returns with flexibility *@loc(this.x)* or parameters *@loc(a,b,c)*. It holds that if every object referenced in the locality annotation is fresh then so is the returned value. Therefore, more methods could be determined as pure based on the more detailed locality information.

Efftp contains other differences compared to JPure such as every method can be annotated other effect annotations to denote which type of effect the methods perform. Efftp provides annotations for throws and IO operations as well. A fresh annotation however is included and freshness is only inferred internally.

The fundamental problem is that Efftp [45] works only on Scala source as far as I can determine which make it difficult to use with the Java source generated with JastAdd. I have failed to run Efftp for Java source in my attempts. It would have been possible to get around this if Efftp worked on the bytecode which is shared by Scala and Java but that doesn't work.

I have failed to run the cache example on Efftp since it's in Java source. Efftps annotations system can however in theory deal with the example. In example 12 is a hypothetical annotation since I have only successfully executed a Scala version.

Example 12: Hypothetical Efftp annotated cache example.

```
@unchecked XType X_value;
@unchecked boolean X_calculated = false;

@mod() @loc() public XType X(){
   if(X_calculated)
       return X_Value;
   X_calculated=true;
   X_Value = X_compute();
   return X_Value;
}
@Pure XType X_compute(){
   ...
}
@Pure XType Y_compute(){
   return X();
}
```

Efftp's annotation system allows for more flexibility than JPure but it is still unclear how arrays are handled. The problem is that it works on Scala source and I can't find any easy way of going over to Java source or binary.

The tools for compiling Efftp was hard to find and dated so it took significant time just

to get Efftp running. There is no obvious problem with the annotation system as far as I understand from the paper [45] if it worked with Java source or java binary. However, I can't really determine since I haven't tested Efftp much since my understanding of Scala is poor. Furthermore Efftp wouldn't be able to check that nonterminal attributes are fresh since there doesn't seem to be any fresh annotation.

Since Efftp won't run on Java code it can't be used as a purity checker for JastAdd.

The annotation system has its advantages over JPure and disadvantages if fresh real doesn't exist. The other extensions and innovations with the *@mods* and *@local* annotations are a reasonable way to deal with more methods and expose more information to the programmer.

5.1.3 Reim

Reim uses a simplification of the Javari type system that deals with reference immutability. Javari was not tested for obtaining purity but Javari is the more advance type system. At least the same purity information could be derived using a similar definition of purity as ReIm [11] the authors of ReIm observed however that Javari pays for its expressiveness by being significantly slower and the Javari inference tool Javarifier[49] is much costlier to run.

Reference immutability is related to purity analysis in that side effects occur when fields are assigned by the wrong pointers. Strong purity would then be equivalent to no assignments and lesser forms of purity would be allowing only certain pointers to assign certain values. An immutable pointer can't change any field's value and only provide read access to the pointed data.

The author for Reim adopts the definition that a pure method is a method such that all its formal and informal parameters and the return parameter are immutable pointers and can be Type annotated as *@Readonly* or are primitive values. The informal parameter is the implicit "this" reference to the current object.

Reim also uses *@Polyread* in polymorphic cases when it might be immutable or mutable depending on calling context. *@Mutable* indicate that a value can be change via the pointer. The Reim variant of the previous example would be to replace *@Pure* with public *@Readonly* XType X(*@Readonly* this.Class this,*@Readonly* ...).

Example 13: Reim cache example.

```
import checkers.inference.reim.quals.*;
public class Reim1 {
        XType X_value;
        boolean X_calculated = false;
        public @Readonly XType X(@Readonly Reim1 this) {
                if (X_calculated)
                        return X_value;
                X_calculated = true;
                X_value = X_compute();
                return X_value;
        }
        public @Readonly XType X_compute(@Readonly Reim1 this) {
```

```
                return new XType();
        }
}


Reim Error:
AnnotatedTests_ReimJavari\ReimSmall\Reim1.java:8: warning:
incompatible types.
X_calculated=true;
^
  found   : @Mutable

  required: @Readonly Reim1
AnnotatedTests_ReimJavari\ReimSmall\Reim1.java:9: warning:
incompatible types.

X_value = X_compute();
^
  found   : @Mutable

  required: @Readonly Reim1
```

This doesn't type check for the same reasons as for why JPure can't handle it. The adopted view of purity is too conservative for our adaptation. To get the example to type check the types would have to be changed so that *X()* has this mutable as shown in example 14.

Example 14: Reim cache example.

```
import checkers.inference.reim.quals.*;

public class Reim2 {
        XType X_value;
        @Mutable boolean  X_calculated = false;

        public @Readonly XType X(@Mutable Reim2 this) {
                if (X_calculated)
                        return X_value;
                X_calculated = true;
                X_value = X_compute();
                return X_value;
        }

        public @Readonly XType X_compute(@Readonly Reim2 this){
                return new XType();
        }

        public @Readonly XType Y_compute(@Readonly Reim2 this) {
                return X();
}
```

```
Reim Error:
AnnotatedTests_ReimJavari\ReimSmall\Reim2.java:20: warning:
call to X() not allowed on the given receiver.

                return X();
                       ^
  found   : @Readonly Reim2

  required: @Mutable Reim2
```

The new error message forces the programmer to change the annotation of *Y_compute()*. The "this" pointer must be declared mutable and the object can be modified and consequently no guarantee on the absence of side effects. As example 14 shows any attribute depending on another, would have to allow any field assignment side effects on the objects involved. Reim is also unable to handle this type of benign side effects.

5.1.4 Javari

The Javari type system specifies a more expressive reference immutability system. The Javari type checker can type check the caching example with the help of increased complexity. Javari adds several new annotations including *@assignable* annotation on field that allows fields to be assigned to even from a *@Readonly* pointer. The example can thus be validly annotated as shown in in example 15.

Example 15: Javari cache example.

```
class Jav1 {
        @Assignable @ReadOnly XType X_value;
        @Assignable boolean X_calculated = false;

        public @ReadOnly XType X(@ReadOnly Jav1 this) {
                if (X_calculated)
                        return X_value;
                X_calculated = true;
                X_value = X_compute();
                return X_value;
        }

        public @ReadOnly XType X_compute(@ReadOnly Jav1 this) {
                return new XType();
        }

        public @ReadOnly XType Y_compute(@ReadOnly Jav1 this) {
                return X_compute();
        }
}
```

Nothing had to be declared as mutable and therefore these methods are considered as pure. Javari can handle this example. However, the protection for the cache fields are

weak and may be assigned anywhere meaning the protection is not fully satisfactory.

Javari can do much more with possibility to declare annotations for each access levels in matrices individually with polymorphism allowed everywhere. This make Javari very flexible but slower and harder to preform inference[12].

A restriction is that Javari doesn't work with object immutability but with only pointer immutability which means the objects can be changed as long as its reachable from at least one mutable pointer. Object immutability could be desired since it gives sufficient side effect freeness guarantee for multi-threaded program to allow thread sharing without having to synchronise the accesses [13].

The Javari type checker only informs about which type a value should have had and doesn't give informative warnings about why. This can be hard sort out what real is the root cause for why the types must be of a given type. This is a drawback shared with Reim. This hinders the ability to give informative warnings.

Javari annotations and the reason for why a reference must be of a particular value can be inferred with Javarifier. Javarifier should be able to return an evaluation sequence that results in that the annotations don't hold. Javarifier runs on a modified version of the Soot compiler which includes a JastAdd made Java parser. I have however been unsuccessful in getting Javarifier's underlying soot compiler to run correctly or to replace it with a new version of soot. The compiler simply crashes with indexoutofbounds or can't load exception. The soot version included with Javarifier fails on Java1.5 source but parses Java1.5. Replacing it has proven difficult. In any case I expect that Javarifier would not implement and address any feature introduced after Java1.5 and this would probably be an convenience even if all the other problem were solved.

Javaris type system is also sophisticated supporting several more annotation that ReIm and without Javarifier it would be too complicated correctly annotate library code and unannounced user code. I would have to implement a checker which determines why the warning was triggered to address both the lack of information in the error messages and the need to annotate code making. This would diminish the usefulness of the Javari.

The added complexity seams excessive for the intended application since unlikely that one would want to separate the different layers of a matrix.

Reference immutability is an indirect way of obtaining purity and doesn't really represent what I want to represent for the application. It's unclear for programmers how to relate them to

There is no abstraction for object freshness so I can't verify that NTA are newly created objects. Therefore, Javari might not be suited for the application either.

5.1.5 IGJ

Object immutability is modelled in for example in the IGJ type system [13] that is heavily influence by Javari. IGJ that apart from having the *@assignable*, *@readonly*, *@mutable* annotations like Javari also have *@AssignsField* which is denoting a method that assigns fields in the current object and *@Immutable* annotations that denotes an immutable object. *@AssignsField* is intended to identify helper methods to the constructor of the object and may be called during construction of an *@Immutable object*. A reference to

an *@immutable* object never be aliased to a none immutable reference. IGJ can handle the cache example using the same annotations as Javari. There is still not any concept of freshness in the annotations so not possible to force a method to return a fresh object.

A further extension to IGJ was also implemented in the Checker Framework [44] than include ownership annotation called OIGJ. Ownership allows restriction of which object may change the value or access the value, relations called owner-as-modifier and owner as dominator. The Ownership abstraction allows creation of immutable lists and separation of the immutability of an iterator and the underlying collection.

The bad news is that Javari, IGJ and OIGJ where are all discontinued and no longer distributed in the Checker Framework. They were removed after version 1.9.13. The type checkers gives as error messages for any side effects they find the assignments at which the type rules is not fulfilled.

5.2 JML

JML is an annotation language that can be used to specify code the pre-and post-conditions that should hold after each methods invocation. JML have lots of annotations for all sort of relation but the ones that seems most interesting for this application is the modifies, assignable and pure annotations. The Pure annotation however is tested too conservative or not at all in every previous statical checkers for JML I checked. I(ESC/Java, ESC/Java2,OpenJML) [50, 43] and related frameworks like JForge [51]. The pure annotation implies modifies nothing.

However, the modifies [fields] and assignable annotation allow some form of reference immutability and region based separation of the heap. That won't be sufficient for ruling out all side effect but a fraction of them could be ruled out. Most JML tools preforms only runtime assertion checks and many of the static checkers uses SMT(satisfiability modulo theories) theorem provers or other model validation prover for example FPV. I have looked at 3 JML tools that could be applied for the application.

5.2.1 ChAsE

ChAsE is JML tool that validates the restrictions imposed by JML assignable clauses, declared using the modifies annotation, on methods. ChAsE doesn't use the full data group scheme[52] which may increase the size of the modifies clauses. ChAsE doesn't test for method purity explicitly and it would not react to side effect from calling unknown code and it doesn't consider aliasing at all. It would however be sufficient to detect many unwanted side effects.

The problem with this type of data group scheme is that the annotation of a method must be a superset of all methods it invokes which means for complicated chains of calls it requires some work to generate the annotations. ChAsE seems to require all the code to be annotated and doesn't use an implicit modify everything as default.

The age of the tool (2002) results in it only tested and built for Java 4 which is before generic, enhanced for loop and all the other Java features introduced under the decade since ChAsE was made. Therefor, ChAsE cannot be used to provide purity checking for JastAdd since ChAsE can't handle code with generics and wild cards which are essential for understand JastAdd code. In JastAdd the use generics starts already in the type declaration for root class for the AST by default called ASTNode.

Its output warning is not the easiest to decipher. Interpreting warning messages on the form exemplified in example 30 quickly becomes difficult for more complicated expressions.

Example 16: ChAsE cache example.

```
class Test{
        private XType X_value;
        boolean X_calculated = false;

        class XType{
```

```
        }
        /*@
         modifies X_calculated;
         */
        public XType X() {
        if(X_calculated)
        return X_Value;
        X_calculated=true;
        X_Value = X_compute();
        return X_Value;
        }

        private XType X_compute() {
        return new XType();
        }

        private XType Ycompute() {
        return X_compute();
        }


}
```

The example would give the warning shown in 16 due to not having also X_value in the assignment clause.

Example 17: ChAsE output.

```
Checking Test.java ...
[[METH1,X]]
Warning: expression
:
:`--= [142]
:   |--X_Value [113]
:   `--( [139]
:      `--X_compute [113]
:
may contain a problem
 Not Passed
[[METH1,X_compute]]
 Passed
[[METH1,Ycompute]]
 Passed
```

5.2.2 ESC/Java2

ESC/Java2 is a static JML checker build as a successor to ESC/Java [53] an even older checker project which terminated in 1996. Thus ESC/JAVA far predates modern Java and doesn't support generics or any but the basic Java features consequently I didn't try ESC/JAVA. It wouldn't run on 64-bit any way. It improved ESC/Java with more JML constructs checked and support for new java features. It has one standalone implementation and one eclipse plug-in version. The standalone implementation doesn't support 64-bit operation system. The constraints generated by the tool are checked with the theorem prover Simplify. The JML modifier "modifies" is supported which can be on the forms "this.*", fields, "arrayname[*]", "/nothing" or "/everything". "this.* " would allow any field in the current object. This could be used to achieve some isolation of the cache fields as in example 16.

Example 18: ESC/Java2 example using data groups.

```
private XType X_value; \\@ in Xdata;
boolean X_calculated = false;\\@ in Xdata;

/*@
modifies Xdata;
*/
public XType X() {
if(X_calculated)
return X_Value;
X_calculated=true;
X_Value = X_compute();
return X_Value;
}

/*@
modifies /nothing;
*/
private XType X_compute() {
return new XType();
}

/*@
modifies Xdata;
*/
private XType Ycompute() {
return X_compute();
}
```

5.2.3 OpenJML

OpenJML is a tool that generates a problem instance to several different SMT solvers for statically testing several JML annotations. The solver however has to be specific version with and the latest versions of both z3 and Cvc are no longer supported and I hade to search awhile before obtaining supported versions. OpenJML is intended as the successor to ESC/Java2. Interesting in OpenJML is an extension to JML for testing of observable purity using a *secret* data group annotation and *query* annotation where fields in a secret data group only can be used by a query annotated function declared to have access to the same data group [54]. The example 28 show a simplified usage of the OpenJML annotations as presented at a conference but the syntax has change in the actual tool. The method X() is allowed to assign field X_calculated because X_calculated is annotated with "//@ in X" which declares the belonging in a data group. This abstraction restrict access to the fields used for caching and the retrieval of the value.

Example 19: OpenJML example using secret.

```
import org.jmlspecs.annotation.*;
class X {

        /*@secret*/ private XType X_Value; //@ in X;
        /*@secret*/ boolean X_calculated = false;
        //@ in X,isCalculated ;

        public /*@ query */ XType X() {
        if(X_calculated)
        return X_Value;
        X_calculated=true;
        X_Value = X_compute();
        return X_Value;
        }
        public /*@ query */ boolean isCalculated(){
             return X_calculated;
        }
        public /*@ pure @*/ boolean z(){
                return isCalculated();
        }
        private /*@ pure @*/ XType X_compute() {
        return new XType();
        }
        private /*@ query */ XType Ycompute() {
        return X_compute();
        }
}
class XType{}
```

Alternatively, since OpenJML also support the assignable and modifies annotations so it could be used as a more modern version of ChAsE. Still the assignable clauses need to

be supersets of all modified location which will be inconvenient and generate complicated annotations considering all fields that JastAdd needs to assign for attribute calculation.

5.3 Type Inferer

Some tools for side effect analysis doesn't seem to have a type checker meaning that they seem to only be able to only infer results but doesn't validate any pre-existing annotations. Purano and JPPA seem to be such tools.

5.3.1 JPPA

Rinard's JPPA [46] is one of the oldest of the mentioned tools with almost all other comparing their results with the JPPA results or at least mentioning Rinards work. JPPA used a whole program "pointer escape" analysis meaning it constructs a points-to graph over all locations pointer can point to which the program has created interprocedurally. By tracing which locations are modified using this points-to graph JPPA can identify both pure methods and immutable references and output a regex for the access path that generate the mutations. A method is not pure if a pointer which may point to memory which "escapes" the method is modified. A pointer escapes the method if it points to location not created by the method.

However, the writers for Purano reported having difficulties to run JPPA in their Java environment [16] and JPPA fails to run most of their test cases. Like the writers of Purano I too have problems run JPPA. In fact, I can't get the JFlex compiler to work on any of my available machines. JPPA did in any case only support up to GNU classpath 0.08 which is far from supporting Java 5 feature or any version of Java. JPPA only supported a tiny subset of the Java standard library.

Even if JPPA could run on modern Java clearing the cache example would require some heuristic which is not included in the approach. Similar later tool like the one made by Geffken does a similar form of analysis but still would require changes to be adopted for this application.

5.3.2 Purano

Purano is a later tool developed by J. Yang and others [16] as late as 2015 which is capable of dividing pure methods into stateless which corresponds to depending only on parameter state and stateful pure methods that also depend on member fields. Purano is built with the intention of preserving the modular checkability of the JPure approach while preserving the traceability of effects present in whole program analysis approaches based on inter procedural checking such as JPPA. To achieve this Purano uses an internally much more sophistically representation of side effects than JPure and Reim and the other type checkers. The annotations store multiple features for each analysed method such as exposed fields or arguments by the return and modified fields and arguments. The tool operates on the byte code representation allowing values to be inferred on library code as well a great advantage over JPure [55].

The tool manages to further relaxes the conditions imposed by analysis from JPure by trying to identify observable pure methods using caching behaviour using 4 heuristical rules and not only white lists [16]. White list is a list of trusted facts which shouldn't be checked.

The cache heuristical are stated in example 20. All four rules needs to hold before declaring the assignments and reads as safe. These rules are insufficiently for JastAdd's parametrised attributes which caches in a map and the rules doesn't allow caching in a collection object. Purano doesn't allow throwing exceptions for a pure method which is a problem since JastAdd throws exceptions for unexpected circularities in the calculations. Exceptions are considered a native side effect by Purano. These short comings can't be avoided since Purano don't provide any way to ignore effects or methods. This is a disappointment since Purano otherwise seemed like a good fit for this application with its implemented detection of cache behaviour.

Example 20: Purano Cache semantic detection heuristic.

```
P1 The field is assigned either by a constant value, or in
only one member function.
P2 The non-constant assignment on the field occurs
within a branch block.
P3 The right-hand value of the non-constant assignment is only
depended on other fields.
P4 The branch condition of the block checks that the value of
the modified member field is a constant value.
```

Purano analysis takes time and for a minimal JastAdd example it takes 30 seconds on full depths since it will read in a large part of the java library even for simple programs. The analysis is interprocedural in its derivation of effects.

Caching the result for the Java runtime is one of the things the authors lists as a possible speedup for Purano. Instead it only generates an inferred result. The annotations are otherwise sophisticated and complicated to manually annotated since they contain much information.

Purano includes new functions to analyse when Purano encounters references to unanalysed methods in currently analysed methods in waves normally until no more new methods are discovered. This behaviour can be altered by limiting the levels of referenced classes to a small number. In which it will only search through x levels of imports. This can speed up the performance greatly reducing runtime down to only a few seconds. At least in any experiment I made limiting Purano to only import two levels did not disturb the result significantly.

A limit of two or three levels would I my opinion suffice for the application since any interesting side effects are likely to have been detected if present by then. For ExtendJ unlimited passes takes 93 seconds and limited to three levels 46 seconds. 93 seconds might be a substantial amount of time for checking on every compilation considering generating and compiling with JastAdd only takes 17 seconds but not prohibitively long time. The result of these analysis is shown in the examples 21 and 22 there the categories stateful and stateless represent caching pure method and fully pure methods. The two-different

setting hardly produces any change in the statistic for the source methods.

Example 21: Purano full analysis of ExtendJ.

```
class 453
method 14918
unknown 0
stateless 2518
stateful 1856
modifier 10544
fieldM 9106
staticM 7612
argM 3686
 [main] INFO jp.ac.osakau.farseerfc.
purano.reflect.ClassFinder
- Runtime :93372
```

Example 22: Purano 3 pass analysis of ExtendJ.

```
class 453
method 14918
unknown 0
stateless 2518
stateful 1861
modifier 10539
fieldM 9101
staticM 7612
argM 3657
 [main] INFO jp.ac.osakau.farseerfc.
purano.reflect.ClassFinder
- Runtime :46085
```

The problem is that Purano doesn't seem to use user annotation or pre-annotated JDK. It thus not possible to ask Purano to verify that a certain sub section of the methods should be pure and give warnings for violations. The analysis also generates 45mb of inferred effect data explaining the effects on each of the 14918 methods. In order to use Purano for the intended application I would have to make a separate tool that processes all this information alongside the Java source to determine if every method associated with attributes is classified as Stateless or Stateful.

N. Ogura, J. Yang did[56] did something similar when they combined a tool that determined which method has changed between two revisions of a software repository and then compared Purano's determination of these methods between the revisions. This finds previously pure methods which have become impure change. I cannot find any copy of their software tool so I haven't tested it.

The data collected by Purano tells you which fields are modified, exposed and which the fields a method depends on but it doesn't tell you which statement that caused this effect. The statement which causes the impurity is information that I would want to provide for any purity warnings. This is a significant shortcoming since generating useful

purity warning is my goal. It also not possible to impose freshness restrictions with Purano. Finally, Purano only works up to Java 7 and if one tries to provide Purano with Java 8 byte code the modified underlying compiler will reject the code due to the Java version. Because of these problems and that it probably would take a significant amount of time it would take to address them, I decided that also Purano is not easily adapted for the application.

5.4 Summary

In summary during testing and research about the different tools a few of the important characteristics are summarised in the tables. The important aspect includes if I could run the tool at all, if it works on a modern 64-bit machine or if they require an older architecture environment. The needed Java version is also checked. If the tool uses Java annotations and JML annotations are also listed. The cache example used through out the chapter is a metric for the tool and if they can type check the code generated for a small JastAdd examples like those in chapter 3.

Table 5.1: Selection of tool characteristics.

Characteristic	JPure	Efftp	Reim	IGJ	ChAsE
Released	2011	2015	2013	2012	2003
Manage to run	Yes	Yes	Yes	Yes	Yes
Works with 64-bit	Yes	Yes	Yes	Yes	Yes
Supports Java 1.5	Yes	Yes	Yes	Yes	No
Compiles/Runs with Java 8	Yes	No	Yes	Yes	Yes
Java Annotations	Yes	Yes	Yes	Yes	No
JML Annotations	No	No	No	No	Yes
Handles cache example	Yes	No*	No	Yes	Yes
Needed no annotation	Yes	No*	No	Yes	Yes
Check small JastAdd example	No	No	No	No	No
Using the which approach					
Pointer immutability	No	No	Yes	Yes	No
Assignable summary	Yes	Yes	No	No	Yes
Freshness information	Yes	No	No	No	No

* Efftps annotation system could clear the example but not the tool.

As can be seen from the tables most tools have a few years since they were last updated. They there for doesn't all support modern functions. If the tool clears Java 1.5 then they tend to compile upto Java 7 with the common reason for failture that then class file version is to high so the dependent compiler can't load the files. This is the case for Purano, OpenJML and Efftp. This limitation applies significantly to the tool which work on Java classes. There reason for this is that most of the tools started to be developed or was developed in the period of Java 7 from 2011 to 2014 when Java 8 was released.

JPure,Reim and ChAsE can be compiled with Java 8 and working on Java source works correctly as long as to complicated Java constructs are used. The indication is to note if I was forced to use an old Java JVM. The Java actually supported is strongly correlated with the tools age and what was state of the art at that time. The Java 5 was released 2005 followed by 6 in 2006 and Java 7 in 2011 and finally Java 8 in 2014. This places almost all of the tools between Java 7 and Java 8 or in the early Java 8 era.

Therefore I indicate that ChAsE supports Java 1.4 but is runnable with a Java 8 JVM.

Table 5.2: Selection of tool characteristics.

Characteristic	ESC/Java2	OpenJML	JPPA	Purano
Released	2011	2017	2006	2015
Manage to run	Yes	Yes	No	Yes
Works with 64-bit	No	Yes	No	Yes
Supports Java 1.5	No	Yes	No	Yes
Compiles/Runs with Java 8	No	No	No	No
Java Annotations	No	No	No	Yes**
JML Annotations	Yes	Yes	No	No
Handles Cache Example	No	Yes	No	Yes
Needed no annotation	No	No	Yes	Yes
Check small JastAdd example	No	Yes	No	No
Using the which approach				
Pointer immutability	No	No	No	No
Assignable summary	Yes	Yes	No	No
Freshness information	No	Yes	No	No

* Purano uses Java annotations but they are currently only placed and manipulated on the Java binary during analysis. Taking the as input would though only be a minor change.

One approach involves immutability annotations for pointers, objects and classes with restriction. The tools Reim, Javari, IGJ and OIGJ followed this approach. The approach can outlaw the modification to attribute but otherwise but the concepts of demanding fresh object from NTAs is not representable which is dissatisfying.

An alternative is to use a secret field and query methods access protection scheme as in OpenJML and similarly in other JML tool using assignable clauses. JML in general has a lot of annotation but hard to find any tool that checks them.

The assignable clauses have the drawback of being hard to combine with the first in a call chain having to allow all locations that all methods it calls assigns. JML has annotations for forcing newly allocated or old objects to be used in different situations among other things which is a strength. JML might propose a lot of annotations but hard to find any tool that test them.

Then there are tools designed with side effect analysis in mind like Efftp for Scala and JPure that uses an own annotation system based on object locality and freshness. These are concepts introduced for reasoning about side effects in a fast and efficient way.

The existing tools are all either too old or probably too sophisticated using powerful SMT solvers aimed towards more complicated analysis and therefore might be too slow to be practically useful for the intended purpose on larger software. This is a further drawback which I find with OpenJML that the communication with the SMT solvers is slow and sometimes confusing. It also forces the use of such a solver.

The question is how expensive analysis are we willing to tolerate. Using a full JML

specification and verifying it is not practical in day to day development. It's also inconvenient introducing more steps translating the Java and specifications to logical theories to prove with the solver.

The problem with JastAdd code is the presence of different situation in which the same attributes needs to be calculated differently. An NTA is an attribute that calculates a new node to the AST and demands in its calculation method to obtain a new node which is problematic if it enlists normal attributes. The helper attributes might be cached and thus might return an old value. A check that an attribute is only every called from a single NTA is needed in this case.

This sort of special casing can't be fully expressed within the type system of any of the tools. JastAdd rewrites are also interesting constructions allowed to perform some modifications but not others.

The circular attributes apart from caching also presents a new special case in which the circular attribute despite caching may call and modify data obtained from itself.

Therefore, I decided to design a new purity checker that deals with these special cases.

In designing the JastAdd and ExtendJ extension I would have to atleast let some aspects of analysis follow the annotations of some of the previous tools, so some form of comparison can be made regardless if the test cases must be compatible with older Java version or not.

6 Proposed architecture and annotations

In this chapter I present an overview of the solution. The intended usage is explained and the annotations system presented. In the next section the more detailed explanation of implementation details is described.

6.1 The specifications for the system

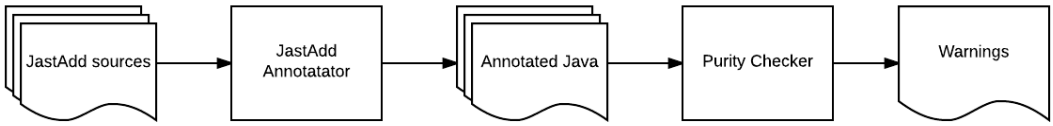


Figure 6.1: High-level overview : Annotate Java and feed code to a purity checker to produce warnings.

In designing a purity checking solution for JastAdd the intended use must be considered. How is work with JastAdd work done?

My purity checker should work on JastAdd generated Java. For attributes, warnings should be generated for side effects. All other methods produced by JastAdd should be allowed to perform its functions without any complaints from the tool.

The checker should be ideally also run reasonable fast and work for large programs. Here a decision had to be made to determine what is consider acceptable. Is the aim for the tool to be run regularly during every stage of development after small changes or in one extensive check only after major changes when lot of changes have been made? If the tool is only intended to be run rarely much more expensive analysis is acceptable and the needs for modular checking would not be as large as if the tool should be used often. If the tool should be used only rarely significantly more powerful points to analysis than Andersen's algorithm like a points-to analysis based on Geffken GAGs[28]. In addition, whole program analysis and extensive inter procedural analysis could be applied despite introducing potential long analysis times.

If the tool should be used often then annotations that are modularly checkable and faster but more restrictive algorithms should be used. Then intended usage of this checker is to be regularly used potentially after every change and thus the checking time should ideally be short. I therefore adapt a modularly checkable annotation system. This annotation system and the checker will incorporate other concepts used in the tested tools in order make the checking process as simple and fast as possible.

Annotations will be used to deal with the problems of recognizing which methods and fields which should be checked and to save resources by not preforming unnecessary checks. JastAdd was modified for this purpose to insert more annotations into the Java code.

There are two main target functionalities for the purity checker. The first one is the side effect analysis. It tests the three segments I listed in section 2 of the JastAdd specification. The goal is to spot potential side effect which would cause attribute values to diverge from their definition.

The other main functionality is to verify several closely related checks. JastAdd wants Non Terminal Attributes (NTA) as fresh objects. Further a NTA must represent a subtree without any null children among other things.

That the checker shall operate on the Java code is clear but the question is which compiler it should be built on. There also is a choice between working with Java source or bytecode or a combination of both. The purity checker I built is designed on top of ExtendJ.

ExtendJ is a Java compiler developed at Lund University using JastAdd. ExtendJ might not provide existing implementations of many of the needed analyses like for example Soot and is also doesn't preform bytecode manipulation. The choice of using ExtendJ is because its developed locally. It provided the opportunity to check how suited RAG is for implementing a purity checker from the ground.

Using ExtendJ as the underlying Java compiler enables the use of RAGs in the specification of the checker. ExtendJ also provides an easy way to work with representation of the Java source. The downside is a checker made only with ExtendJ would not be able to easily check any library code.

To handle library code with a checker that can't read Bytecode an efficient annotations system is needed. The annotation system must be expressive enough to allow the specification of the effects of the most common library functions. The annotation also should be simple enough for the users to write and understand since they would have to.

6.2 Solution overview

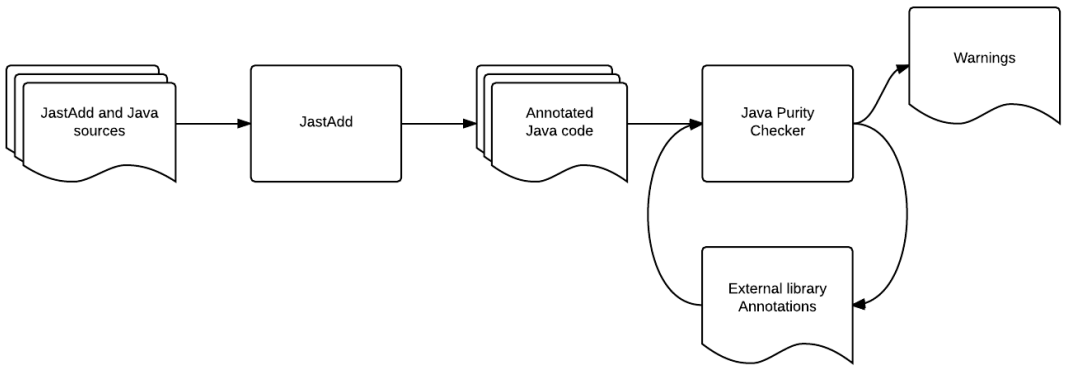


Figure 6.2: Solution overview: JastAdd generated annotation + manual annotations run through the checker to generate warnings.

The work process for producing purity checked programs using JastAdd is described in figure 6.2. From writing the JastAdd source code to purity checking the program.

The work process for the tool developers change slightly. It gains some extra steps.

1. Produce JastAdd specification normally
2. Annotate plain Java portion of the specification
3. Apply the purity checker to see if library methods need to be annotated
4. Inspect the generated annotations for the libraries, and correct them if needed.
5. Iterate steps 4 and 5 library methods annotations are needed
6. Iterate until no more method are annotated.
7. Apply purity checker to see if there are detected problems in the JastAdd specification
8. Modify specification based on the detected problems
9. Iterate steps 7-9
10. Purity checked program without any detectable problems

The developers write equations and attributes normally and the modified JastAdd compiler will generate annotations for them. This will automatically take care about the method annotations for any attribute used independently. The developers simply should take care about any annotation for the normal Java methods, attributes which are used helpers for NTA calculation and finally of the parameters which the needed different that standard behaviour.

However, Java methods which are either invoked during attribute calculation or which independently needs to be checked must be manually annotated since these are not annotated by the JastAdd tool. The annotation need to be placed on both methods and parameters to allow or disallow different types of side effects.

The correctly generated annotated Java is then feed into the purity checker. The source for all methods used doesn't need to be available or parsed for the purity checker. Library code for which the Java source is not available must be described in one or more property files with library annotations. The generation of library annotations is work that needs to be performed once. Once the annotations for the commonly used library functions such as usage of the different forms of collections have been synthesised it can simply be used for all projects.

When generating library annotations, the checker tries to suggest a level of purity needed to avoid the warnings but is not sophisticated enough to generate annotations that doesn't need to be manually tweaked. For the generated warnings to be correct the list of annotations needs to be manually verified and corrected. Depending on the corrections provided to the annotations more methods might be needing annotations. This means that step to generate library annotations is iterative.

The checker can separate the suggestions into two categories the annotations for the actual library methods and methods in the provided source code that needs to be annotated. The annotation generation should then be run again to see if the new annotations or any manual changes causes more methods needing annotation to be detected.

Finally, the purity checker is then used to generate the warnings and based on the results the program source is modified to correct the problems if any were detected. When program source code is later modified its rerun to check if any new problems has occurred.

Purity annotator

The Java annotations for the attributes is generated with JastAdd. JastAdd knows what needs to be checked and what's part of the internal machinery. I thought that this knowledge should be used.

The use of Java annotations allows the checker to be used independently of JastAdd in Java programs. Java annotations are already supported in Java and existing support for checking correct placement is built-in the Java compiler. In Java 8 the definition of annotations was extended to allow annotations everywhere where types are used [57] where it previously could only be placed on declarations. The difference is shown in example 23

Example 23: Annotations in Java 7 vs Java 8.

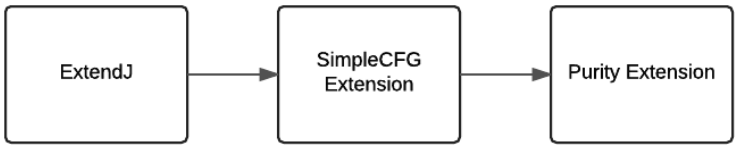
```
// Java 8 SE Annotation
@Annot List<@Annot A> x =
(@Annot List<@Annot A>) new @Annot List<>();

//Java 7 and ExtendJ only allows
@Annot List<A> x = (List<A>) new List<>();
```

Despite that ExtendJ only currently supports Java7 annotation style. This is still sufficient for the intended purity annotations. The Java annotations can be made to suit the needs for a purity checker and no extended special annotation schema are needed. Using the Java annotation over a special annotation saves the work to implement a new annotation schema. Any conflict with any other tools that might be used in combination is also avoided with Java annotation. Custom annotations schemes could conflict with other tools that might want to be used in combination with JastAdd such a JML tools.

The inability to place annotation everywhere where comments as would be allowed by JML styled annotations would only have minor effects for the checker and the workaround are not complicated. The purity checker acts on declarations and the effects works with each variable as a unit. This means that for example ignoring problems or imposing some condition for only a particular assignment of a variable is not possible. The analysis is done for all or none of the usages of the variable but I consider this a minor inconvenience at most.

Purity checker

Figure 6.3: Purity checker construction.



The purity checker is built on top of ExtendJ [17]. For purity analysis, the information about if objects are newly created is significant and need to be propagated along. Jesper Öqvists SimpleCFG [58] extension which provides control flow analysis could be used to propagate this information along with non nullity and other facts. The purity checker would therefore be built on top of SimpleCFG [58] extension or part of it. The figure 6.3 illustrates this. I choose SimpleCFG over Intraflow because the representation is sparser than Intraflow that's more expressive than necessary. SimpleCFG is compatible with the latest versions of ExtendJ which Intraflow is not.

The SimpleCFG extension provides a control flow graph representation with easily changeable granularity. SimpleCFG represent the CFG as a graph of new nodes and connects to the original AST.

The default configuration of SimpleCFG represent branches, method calls, exceptions and return statements. To do purity analysis, I needed to represent object assignments as well. The CFG representation was therefore modified to include assignments and variable declarations which are points of interest for the purity analysis. The assignments and declarations are where it determined how later usage of the memory location should be interpreted.

6.3 Annotations

For the extension, I introduce some new annotations on methods, fields and parameters. The chosen annotation scheme is inspired by the different tested tools combining ideas from OpenJML, Efftp and JPure along with some own extensions. The choice of JPure as basis is due to the annotation being reasonable suitable for the problem and that they have been previously extended. The JPure annotations as basis allows comparisons with the analysis from JPure and JPure's test suit can server as starting point for the work. The same close comparison is not possible with the other tools. I only borrow ideas and parts of Efftp and OpenJMLs system and not the full system.

From OpenJML the idea of secret data groups for handling parts of state that should not be considered part of the observable state is carried over. The introduced annotations are complemented with the pre-existing annotations that JastAdd is already using.

There are 8 annotations introduced by the extension. The introduced annotations are *@Fresh*,*@FreshIf*,*@NonFresh*,*@Pure*,*@Local* *@Secret*,*@Entity* and *@Ignore*.

@Fresh,*FreshIf*, *@NonFresh* are the freshness annotations and can be placed on both methods and parameters. They impose restriction on the returned object when placed on method and restriction on the passed argument when placed on parameters. The others except *@Entity*, are purity annotations restricting or allowing side effects.

The full description of where the annotations can be place are shown in the table below. The methods represent any form of method call and there is no difference between constructors and any other methods as far as applicable annotations.

Annotation	Type	Methods	Parameters	Fields	Locals	Types
Fresh	Freshness	Yes	Yes	Yes	Yes	No
FreshIf	Freshness	Yes	Yes	No	No	No
NonFresh	Freshness	Yes	Yes	No	No	No
Pure	Purity	Yes	No	No	No	No
Local	Purity	Yes	Yes	Yes	Yes	No
Ignore	Purity	Yes	Yes	Yes	Yes	No
Secret	Purity	Yes	No	Yes	No	No
Entity	Behaviour	No	No	No	No	Yes

Purity Annotations

The four "Purity" annotations can also be parametrized with a group property for which data group the method belongs to. In the case of Secret the data group is declared. Also *@Fresh* may be parametrised with a data group.

@Secret(group="group") declares a section of the program state unobservable for all methods except the methods with the correct data group. This is the syntax used in OpenJML. JML datagroups syntax could equally well have been used by enabling ExtendJ to parse JML annotations. The intended usage however doesn't necessitate the full expressiveness of the JML assignable clauses and datagroup concepts.

@Pure(group="group") declares a method allowed to access and operate with the state hidden with *@Secret(group="group")*. The intention is that Secret on all cache fields

would outlaw any method from doing the error of directly trying to read a cache field. The cache should only be reached from the using method.

In the JastAdd case it could also protect the calculation machinery. As shown in the example 24. An unparameterized *@Pure* doesn't access any hidden state.

Example 24: Annotated simplified Java for an attribute.

```
@Secret(group="beta()")
protected int beta_visited = -1;

@Secret(group="beta()")
protected boolean beta_computed = false;

@Readonly @Secret(group="beta()")
protected int beta_value;

@Ignore private void beta_reset() {
  beta_computed = false;
  beta_visited = -1;
}

@Pure(group="beta()") public int beta() {
  ASTNode$State state = state();
  if (beta_computed) {
    return beta_value;
  }
  beta_visited = state().boundariesCrossed;
  beta_value = beta_compute();
  beta_visited = -1;
  return beta_value;
}

/** @apilevel internal */
@Pure private int beta_compute() {
  return 2;
}
```

In the example, the calculation method "beta_compute()" which contain the user code cannot assign or read the secret fields beta_visited, beta_computed or beta_value. These fields are used by the JastAdd internal machinery for evaluating attributes. Because of the annotations, they can't be read by any user code by mistake. Use of Unique groups of each attribute would provide optimal protection against any access of field used internally by the JastAdd system. Programmers who understand how JastAdd is intended to be used however would not make such mistakes therefore unique groups are not really necessary.

The annotation *@Ignore* on a method means that any side effects inside should be completely ignored and not considered by the checker. It should be used very carefully to simplify the checking process and allow resetting of hidden state. Any other annotations on the method or parameter must however be fulfilled. The @Ignore annotation is useful for cases when the analysis is not strong enough, and reports an error when there is no

error. In the above example it is used for resetting hidden state. Another use of @Ignore is to annotate a method called ”flush” that clears the attribute caches.

Example 25: Example signature with @Ignore where the methods them self are not checked but the passed parameters are.

```
@Ignore public Object a1 (Object b, Object c);

@Ignore public Object a2 (@FreshIf Object b, Object c);

@Ignore @Fresh public Object a2 (@Local Object b, @Local Object c);
```

The annotations *@Fresh* and *@Local* fill essentially the same roles as in JPure. *@Fresh* denote that the method returns an object that didn't exist prior to the methods execution and *@Local* is for local changes.

The novel annotations *@NonFresh* and *@Entity* are extensions specifically for JastAdd and similar applications where we must assure that objects of certain classes for example in the JastAdd case are not created and returned unless in a specific context. The *@Entity* on class denotes such a class that may not be created unless in a specific context. *@NonFresh* denotes a method that opposite to *@Fresh* annotated methods are guaranteed to return an object that is not Freshly created.

6.3.1 @Secret

The *@Secret(group="")* annotation is based on the same annotation in OpenJML and other data group schemes. It defines a section of the object state that should have restricted visibility such as caching or hidden debug counters etc. The group restriction can protect a field to only a single method if that is necessary often it might be sufficient to only hide the information.

The important detail is that assignment to a secret field should not be considered a side effect. Local changes by which I mean the changes to same object and other considered part of the same state are also allowed. In the section about *@Local* I explain in detail what's considered a Local change. The reason for this to allow caching of many values in a map by necessary requires the map to be modified.

- A field or method can be annotated *@Secret(group="group")* to restrict access to them.
- A Secret field can only be read or written to by a method annotated with a purity annotation with the same data group or a *@Ignore* annotation.
- The intention is that a secret field should only be changed directly from one method and otherwise should cache immutable results or data never exposed to the end user.
- An object should only change the secret state of itself and not allowed to change a different *@Secret* field access via a *@Secret* field.

Example 26: Example use of Secret.

```
public class Secret{
    @Secret(group="group1") boolean b;
    @Secret(group="group1") List<String> secretlist;
    @Secret(group="group2") int a;
    private int open;

    // May not change the Secret fields
    public Secret(int x){
    open=x;
    }
    // May change group2 fields and un annotated
    @Pure(group="group2") Secret(int x,int y){
    open=x;
    a=y;
    }

    @Pure(group="group1")
    public List<String> worker(){
        if (b)
        return secretlist;
        secretlist=calcSecret();
        b=true;
        return secretlist;
    }
}
```

In the example 26 a small class is shown. There are three fields that has be put in two secret groups. The method "worker()" is allowed to use the group "group1" for caching the list. After completion, the produced list may not be changed unless by an impure method or an @Ignore annotated methods.

Only in very specific situations may a cached value be changed after it creation method. This applies to a few exception scenarios for JastAdd regarding circular attribute evaluation where the same attribute might revise its own value and NTAs constructed with a helper attribute. In the case of helper method for NTA creation it may only be called as part of the NTA creation. More about these special cases when I apply the annotations to JastAdd code.

6.3.2 @Pure

Only methods can be annotated *@Pure* to declare that it should be observable side effect free and checked. There are the parametrised variant *@Pure(group="group")* which gives access to manipulate a secret data group.

A *@Pure* method is checked by the checker to see that no side effects is introduced except for changes to the secret state.

This means a method annotated *@Pure* should follow the following restrictions

- Doesn't assign fields unless part of secret datagroup.
- Doesn't modify any reference pointed to by a parameter unless annotated @Ignore or @Local.
- Doesn't modify any fields on anything but known newly created objects.
- Doesn't call any method including constructors which are impure.
- Doesn't call any method including constructors annotated @Local unless called on a newly created object.

Further changes compared to the *@Pure* of JPure is that the method can have *@Pure* while the parameters have *@Local*. The annotation placed on the method apply to the changes made to caller object "this". A *@Local* annotation on the parameter would allow the local change to the parameter passed object. This allows two different ways of expressing the same level side effect freeness as shown in example 27 while in JPure it was not possible to separate to only allow change to the parameters without the caller.

Example 27: Example two set of equivalent method signatures in this annotation system.

```
//May change this,b
@Local public Object a1 (@Local Object b);
public Object a1 (@Local ThisType this,@Local Object b);

//May change b but not this
@Pure public Object a2 (@Local Object b);
public Object a2 (ThisType this,@Local Object b);
```

6.3.3 @Local

The *@Local* annotation declares that the method induces "local changes". A change contained within a particular object or object chain created together. This means assignment to the objects fields. Placed on the method it means that the calling object may be changed. Placed on a parameter it allows local change to the parameter object.

The meaning on fields is slightly different by combining the state of two objects into one bigger abstract state called a locality by D. Pearce in JPure. I will use the terminology here since my interpretation of *@Local* on field is equivalent.

The *locality* of an object is all the program state that should be considered part of the same "object" as far as the checker is concerned. All the state in a locality shares as single freshness status. The locality contains all the fields in the object but for *@Local* annotated fields the referenced objects' locality is also part of the locality. This makes multiple objects effectively part of the same state.

The shared freshness allows the checker to know that all the objects in the locality are fresh together. This meaning the referenced objects of all fields annotated local are also fresh if the parent object is fresh. In this way *@Local* allows changes which may span the fields of several objects with in a specific subtree.

A more accurate analysis would have to deduce and share the freshness status of each field in an object between methods. For JastAdd I thought this would be unnecessary. In the most common situations the freshness is either always fresh or never and this can be resented with a annotation.

Apart from being allowed to change annotated variables locality, a *@Local* method must be *@Pure* in all other aspects. The method must be side effect free and meet the requirements for *@Pure*.

There are two extra rules for what type of change are allowed introduced by Pearce to allow for simpler analysis that's sufficiently expressive for most scenarios. The objects assigned to *@Local* field must also be newly created for successive modifications to remained confined to the same state as was allowed to change at the method start. The fields not annotated thus serves as end points for the locality. The end point fields can be assigned any value since that value will not be included among the modifiable. In other words the end point field represent the last assignable location in a access chain.

A local method does only impose limited side effect on a limited number of objects. These can thus be safely applied to fresh object without changing the freshness status of the object. It is still Fresh.

Example 28: JPure @Local rules.

```
Definition 3 (Local Method).
A local method may modify the locality of any parameter
Annotated @Local but, in all other respects, must remain
pure. The method receiver (i.e.this) is treated as a
special parameter, with @Local placed on the method itself.

Rule 1
A local method may assign fresh objects to any field
in the locality of a parameter annotated @Local.

Rule 2
A local method may assign any reference to a field
in the locality of a parameter annotated @Local
provided that field is not itself annotated @Local.
.
```

Extensions to the concept is introduced when we consider the constructs beyond Pearce consideration. Arrays, List and collections are constructs with can contain inner objects. Arrays needs to be considered alongside variable arguments since they are represented the same. "*@Local* X..." might be interpreted as short hand for "*@Local* X a,*@Local* X b,..." for the programmer but as "*@Local* X[]" by Java which is different under the Pearce rules.

The first clearly imply that all the elements are also local but the other only specify the field storing the array should be covered by local. The *@Local* on an array means only the field storing the array is changeable in JPure. I mostly adopt the same rule as In JPure with a *@Local* annotated array not necessary containing only objects we may change. The inner elements will be conservatively estimated by the checker when modifications occur or an inner element is return as something that should be fresh.

The inner elements are only guaranteed to be included for a *@Fresh* annotated array which can then only contain fresh objects. The correct translation in my implementation is that *@Local* X b,..." accepts "*@Local* X a,*@Local* X b,..." and *@Fresh* X[] b".

There is one parameter that can be provided with the annotation when placed on methods elsewhere the parametrised versions are meaningless. The annotated version *@Local(group="a")* allowed to access fields belonging to the secret datagroup "a".

6.3.4 @Ignore

A @Ignore annotation is intended to silence warnings. When placed on a method it silences the warnings for anything happen inside the method. A method annotated @Ignore is thus allowed to purge caches but it the programmers job to make sure that if a method is annotated with @Ignore it won't leave undesired observable side effects.

When placed on a variable declaration or parameter any warning triggered by these variables will be silenced. Any value from an ignore variable has the freshness that necessary.

For the most part shouldn't end users use the Ignore annotation except to silence some purity warnings in situation when the analysis can't sort out some benign side effect that the programmer trusts.

The effects of an *@Ignore* annotation are summarised in the following list.

- A method can be annotated @Ignore to state that the analysis should ignore side effects in the annotated method.
- A method annotated @Ignore may change any field even those annotated @Secret. This apply not only to the own object but to any reachable object.
- Any method regardless of annotation may call a method annotated @Ignore provided that any other annotation on the method and its parameters are fulfilled.
- Any method annotated @Ignore returns an object of whatever locality is necessary for the situation unless annotated otherwise.

- @Ignore on a parameter or variable means that all side effect induced on that object are ignored.

It's mostly only intended to be used by the JastAdd system to avoid preforming unnecessary checking work of the JastAdd internal methods which are trusted apart from cache purging. It's applicable to all the methods of the JastAdd system that's not supposed to be exposed to the developer.

An *@Ignore* annotation is not used by any of the mentioned tools but an adaptation for the application. It helps dealing with the cache issue that needs to be both protected from access and purge able. Furthermore, it's unreasonable to assume that the tool will allow all the code, a programmer would want to be allowed to use and thus some time the programmer would want to silence warnings.

Freshness Annotations

The freshness annotations are concerned with what object a method returns. The question whether they are newly created, definitively not a new object. This information is needed to determine if the returned object may be changed by the receiver.

The annotations states if the returned value is fresh or nonfresh and under what conditions. Absence of annotation means unknown freshness and the returned object may be either fresh or non fresh. The *@Fresh* and *@NonFresh* specified unconditional known freshness of either fresh or non fresh. In combination with the conditional freshness from *@FreshIf* a freshness which depends on caller and parameters can be determined.

6.3.5 @Fresh

The *@Fresh* annotation is declare on a method that returns newly created state. If the method is not otherwise annotated its implicitly annotated *@Pure* and must fulfill the requirements to be annotated *@Pure* in addition to return an object newly created within that method. The annotation can be combined with *@Local* meaning the concept of freshness is more independent of the method purity than in JPure. The *@Fresh* in JPure is inseparable connected with method purity implying *@Pure* and I have relaxed this to implying at least *@Local*.

To be considered fresh the object and its locality must be Fresh meaning any object in *@Local* annotated fields must also be assigned newly create objects or left as null. Locality can here be used to define which subtree of objects accessible which must be newly created at the same time and which that doesn't. The freshness of an object tells the purity checker that local manipulations can be made to the object without introducing side effects.

@Fresh is desirable to have for JastAdd in certain situations that requires a newly created AST subtree. This applies NTAs[59] where the equation must produce a new node to the AST with all children initialized and newly created. JastAdd furthermore want them to be nonnull. Nonnull is not built in freshness requirement per default but enable in combination with special rules for *@Entity*.

- A method can be annotated *@Fresh* only if it returns a newly allocated object with a newly allocated locality.
- A method can be annotated with *@Fresh(group="group")* with the same effect as for the purity annotations with a group annotation.
- The method annotated *@Fresh* must also satisfy rules for *@Pure*.
- A method can be annotated with the parametrised *@Fresh(group="group")* to also be able to access the secret data group "group".
- A parameter can be annotated *@Fresh* if it must correspond with a fresh object. This is equivalent or stricter with *@Local* and the difference is in regards to arrays.

6.3.6 @FreshIf

@Freshif is a conditional freshness indicator. A method with any *@FreshIf* is only returning a fresh object if all parameters annotated *@FreshIf* are fresh and in the case of *@FreshIf* on the method also the caller. The *@FreshIf* annotations on a method signature serves a similar function to how *@Local* was used in Efftp. It provides additional capabilities to define the behaviour of the methods that the checker can't observe from the other annotations.

The examples in example 29 illustrates the additional information provided by *@FreshIf*. A method with *@Local* doesn't need to return a fresh object and might not be called from a fresh object due to secret special cases. In cases where source is available cases such as the exemplified can be discovered with little analysis but for library functions that option is not available. With *@FreshIf*, *@Fresh* and *@NonFresh* the annotation system has complete representation of the freshness status.

The last example illustrates a special case for collections and maps which also effects arrays. It's occasionally useful to store a collection of fresh object in a container object to keep track of a number of object for which a some operation should then be applied. In JastAdd this corresponds to a calculation approach JastAdd uses for collection attributes. I demonstrate an example of that in section 8 when I discuss collections attributes.

Example 29: Example usage of *@FreshIf*.

```
import java.util.Set;

public class Test{
   int y = 9;
        @FreshIf @Local public Test freshIf(){
                return this;
        }
        @Fresh public Test freshIf2(){
                return this;
        }
        @Local public Test freshIf3(){
                return this;
```

```
        }

        @Fresh public void freshIf3(){
                Map<Test, Set<Test>> x= new ...();
                x.put(this,new HashSet<Test>());
                modSet(x.get(this));
                // Can only work under
                // @Local Map<...,...>.put(@FreshIf x)
                // @FreshIf Map<...,...>.get(this);
        }

        @Local public void modSet(@Local Set<Test> t){
                t.add(this);
        }
}
```

@FreshIf is not necessitated by the demands from JastAdd but I thought it was useful to introduce for the potential usage in user code and the simple check it would .

6.3.7 @NonFresh

The *@NonFresh* annotation is declared on a method that is guaranteed to return an object that existed prior to the method execution. It can also be placed on parameters forcing the parameter to not be a newly created object.

This annotation guarantees that the value doesn't satisfy *@Fresh*.

6.3.8 @Entity

The *Entity* annotation declares a class and all it subclasses as an entity type. This changes the default assumptions about objects of these types. An object of entity type is assumed *@NonFresh* if unannotated instead of maybe fresh that is the default assumption for not annotated parameters and methods. A method returning an Entity type are implicitly NonFresh unless overridden by a *@Fresh* annotation.

The freshness analysis get modified expectations and want that entities assigned to *@Local* fields should be non null otherwise null is allowed as a fresh value. Further more Entity

Entity are treated differently regarding how the are allowed to be created. Invoking a "attribute" annotated method on an entity type which is newly created is not allowed.

A *Entity* type must be created in an "nta environment" for methods with the "isNTA" property for the JastAdd annotations.

JastAdd annotations

JastAdd already generates some useful annotations. The annotations include "@ASTNodeAnnotation.Attribute" and "@ASTNodeAnnotation.Token". These are placed on the methods which obtains the attribute or the "Token". The purity checker uses these

annotations to improve warning messages and to adapt the analysis for some kinds of attributes.

A Token is a value set by the parser during the tree construction. A Token may not be set in any Attribute according to JastAdd reference manual[6] since they have side effects. The ASTNodeAnnotation.Token doesn't have any significant role for the checker apart from indicating a value that may not be indicated to be fresh and thus editable when obtained. It's used to enhanced the warning messages slightly. An example of this is illustrated in example 31.

Example 30: Example of enhanced messages.

```
Enhanced warning message:
The method call A.getX obtains an AST token. Tokens cannot
be considered newly created and changeable after initial
assignment by the AST parser.

Normal warning message:
The method call A.getX does not return a newly allocated object.
```

@ASTNodeAnnotation.Attribute has several arguments showing what sort of attribute the method represent of which two have some important for the checker. The arguments "isNta" indicating that it creates an NTA and "isCircular" for any attribute that might depend on itself indirectly. These arguments are also used to enhance the warning messages by displaying a few effected attributes by a problem. The "isNTA" and "isCircular" properties is used to identify that the behaviour has to be tweaked from the normal attributes.

Example 31: NTA using a helper attribute. The helper attribute needs to be considered fresh. This is determined with help of the isNTA annotation and a call graph check.

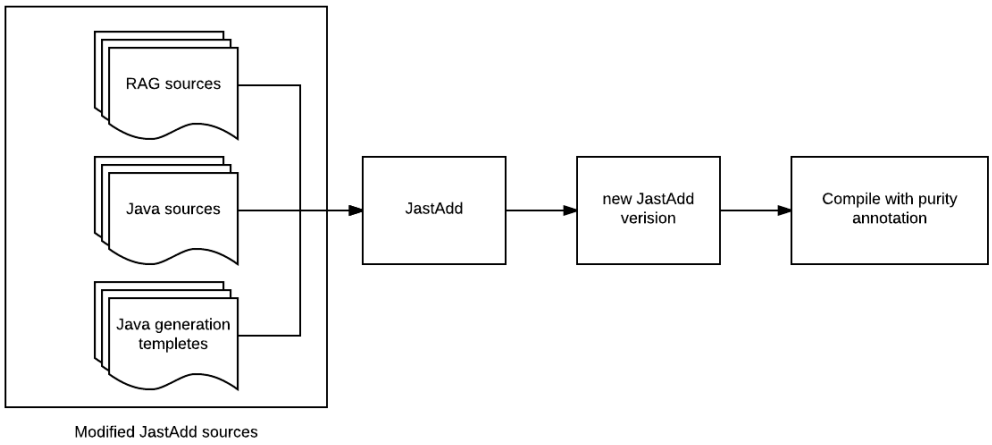
```
//nta using a helper attribute
nta A ASTNode.getA(){
        int i=5;
           return creation(i);
}
syn A ASTNode.creation(int i)=new A(new B(ID), new B(ID));

// Generated code
@ASTNodeAnnotation.Attribute(isNta=yes,isCircular=no)
@Pure(group=_ASTNode) public A ASTNode.getA(){
            ...
        node = compute_getA();
        return node;
}
@Fresh protected A ASTNode.getA(){
        Int i=5;
            return creation(i);
}
@Pure(group=_ASTNode) public A ASTNode.creation(int i){...}
```

7 Solution Implementation

In this section, a more detailed exploration of the solutions implementation for both tools are given and some motivations for why the solution is designed the way it is. First the purity annotator version of JastAdd is explained and my purity checker which I have called *ExtJPureChecker*.

7.1 Purity Annotator

Figure 7.1: Constructing a purity annotating JastAdd variant.



The annotations are implemented as Java annotations and not as keywords for either JastAdd or *ExtJPureChecker*. Since the annotations are Java annotations the annotations can also be used for ordinary Java programs not generated by JastAdd.

The first change made to JastAdd was to let JastAdd generate a Java class ”SideEffect.java” that contains the declarations of the annotations to be used in the analysis, e.g., @Pure. This SideEffect class has the same package as the AST classes. Hence, these classes can refer to the ”Pure” annotation as *SideEffect.Pure*.

The class SideEffect could be imported in order to allow writing unqualified annotations.

Example 32: The side effect annotations generated by JastAdd.

```
/**
 * @ast class
 * @declaredat ASTNode:259
 */
public class SideEffect {
        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target({METHOD,
                CONSTRUCTOR, PARAMETER   })
        public @interface FreshIf {
        }

        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target({METHOD,
                CONSTRUCTOR, PARAMETER,
                LOCAL_VARIABLE,FIELD   })
        public @interface Fresh {
                String group() default "";
                boolean rewrite() default false;
        }

        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target({METHOD,
                CONSTRUCTOR,PARAMETER,
                LOCAL_VARIABLE,FIELD   })
        public @interface NonFresh {
                String group() default "";
        }


        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target({FIELD,
                METHOD   })
        public @interface Secret {
                String group() default "";
        }


        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target({TYPE   })
        public @interface Entity {
        }
```

```
        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target({METHOD,
                CONSTRUCTOR,PARAMETER,
                LOCAL_VARIABLE,FIELD   })
        public @interface Ignore {
        }



        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target({METHOD,
                CONSTRUCTOR,FIELD,
                PARAMETER   })
        public @interface Local {
                String group() default "";
        }

        @java.lang.annotation.Retention(
                java.lang.annotation.RetentionPolicy.RUNTIME)
        @java.lang.annotation.Target(METHOD,
                java.lang.annotation.ElementType.CONSTRUCTOR)
        public @interface Pure {
                String group() default "";
        }
}
```

As displayed the changes to JastAdd simply consist of inserting the correct annotation into the right template. In most template, the change is to place *@Pure* with secret datagroup for the attribute returning method and without data group for the compute method. The exception is NTAs where the compute method is *@Fresh*.

All the default AST constructing methods have also been annotated. The deep internal methods with a combination of *@Ignore* and annotation on what they should be passed. That is to save time and not analyse none user code too much. An example is "@Local setChild(@Fresh ASTNode)" which is called from the constructor of an ASTNode which demands that each child to an ASTNode is a newly created ASTNode.

In a few cases the method signatures and cache fields is generated entirely internally in the JastAdd code this applied to for example rewrites and circular collections. These also needed to be annotated and required a tedious manual search of the JastAdd code to find.

7.2 Purity Checker

The purity checker has two tasks. It's supposed to collect all the warnings for when side effect occurs by checking the annotation and display them to the programmer. Secondly it

supposed to generate the annotations for external libraries and Java source not annotated by the annotator. The JastAdd annotator could have partly provided these annotations during interweaving but I didn't have time to study JastAdd enough to provide this functionally, I spent my time with programming the checker in ExtendJ. They therefore have to be inferred by the purity checker.

The second task depend on the first one. Warnings trigger because methods were insufficiently pure or had the wrong freshness annotations so the needed annotations depends on which warnings are triggered. In the case of external libraries, the are no annotations present in the start meaning they all default as impure. The methods that should be side effect free will thus issue a warning for any usage. To avoid the warnings, the correct annotations must be determined and for this the checker first collects all warnings and then controls each of them to determine the annotations it demands. The checker is helped by which JastAdd annotations are present to more precisely deduce the needed annotations but they are still mostly suggestions that will need manual corrections.

The warning could have been triggered because of a parameter was lacking an annotation or the caller was modified and in these cases the checker will suggest that the relevant part of the method signature be annotated. This inference of library method annotation does not have access to the Java source of the method and can thus only check the signature. but even for the methods with Java source I only infer for the signature.

The annotation inferences can only make suggestions for the method signature and field annotations. It cannot suggest for any local variables. The reasons include that it would be more difficult to check and for the most part the local variables shouldn't have any special behaviour.They should mostly be treated as genuine errors. I therefore haven't prioritised that and then I haven't had time to include inference for that.

Types are not annotated beyond what my JastAdd annotator provide which only annotates the AST classes as entities. This enforces the type to be considered special and imposes some JastAdd related restrictions. I leave it up to the programmer to specify if any other class needs the restrictions I described in the section 6.3.8. Finally, I don't fully determine fields to annotate with *@Local* and impose the associated restricts and neither for the *@Fresh* and *@Secret*. Fields the programmer manually uses as cache fields could be annotated with *@Secret* according to the rules used in Purano [16] but I don't currently provide this. The JastAdd annotator can only provide annotations for the field which are part of attribute calculation. There are thus several parts that must be annotated manually.

Example 33 shows a small section of an annotation list. It is constructed to be written and read by the Java properties class [60] which builds on a hash table . The format consists of key value pairs with first a complete method signature specified by package followed by "=" and then purity information. The list specify method purity annotations which consist of annotations for the methods and their parameters where each may have multiple annotations. To specify all information related to a method with only one key I store the information combined into one string. The string can then be split using "-" and "," such that "," denotes more annotations and "-" moves to the next parameter to recover the annotation information. For example

"java.util.stream.collect(java.util.stream.Collector)=Local,FreshIf-FreshIf" would indicate method has *@Local* and *@FreshIf* and the parameter *@FreshIf*. Under this annotation we can deduce that the return from this method is only fresh if both the parameter and the caller is fresh. An more concrete situation demonstrating how well behaviour can be specified is shown in example 34 where the correct annotation is "Pure,FreshIf-FreshIf-Local" which encapsulates the behaviour completely.

Example 33: Example list of annotated methods of library methods.

```
#Wed May 03 08:44:02 CEST 2017
java.util.HashSet<T>.<init>
(java.util.Collection<wildcards.?\ extends\ T>)=Fresh
java.util.HashMap.<init>(int)=Fresh
java.util.Collections.singleton(java.lang.String)=Pure
java.util.Set<T>.remove(java.lang.Object)=Local
java.util.Set<java.lang.String>.add(java.lang.Object)=Local,FreshIf-FreshIf
java.util.stream.Collectors.toSet()=Pure
java.util.stream.collect(java.util.stream.Collector)=Pure,FreshIf--
java.util.Collections.emptySet()=Fresh
Test.X.trickyFresh()=Pure,FreshIf-FreshIf
```

Example 34: Example motivating trickFresh()

```
Int k;
public X trickFresh(X a,X b){
        b.k++;
        if (a.k > this.k)
                return a;
        return this;
}
```

I have made the checker such that it takes previous lists of annotations and based on those and annotations in source code suggest new annotations separated into two different lists. One list for annotations needed in the source code and one for external methods. Java properties [60] are used with the method signature as key and the annotations for method and parameters separated with "," and "-".

Since the design of the checker is to modular check the methods. Any internal side effects apart from in a few special cases can just be collected with a collection attribute that visit the AST for the method and check if any assignment, call or read causes any problem.

The first type of warning that the checker identifies is invalid annotation combinations both for newly parsed information and for the library annotations.

Type rules

Not all combinations of annotations are allowed and for inconsistent and redundant annotations warnings are issued. The Java compiler will, due to the "target" property on the annotations, reject some combinations. For example, the @Secret annotation must only be placed on fields and cannot be placed on types or methods. This can

be guaranteed by the Java compiler. The Java compiler will also rule out redundant duplicate annotations by following the annotation rules.

This leaves for the checker to simply verify that of the applicable annotations the given combinations are consistent. The annotations must be consistent in the class hierarchy as well.

In the table below are all consistent annotations for method placed. Only one freshness annotation is allowed (Fresh, FreshIf, or NonFresh), and it can be combined with Local or Ignore. Pure can only be combined with FreshIf but it is implied every time a freshness annotation is used unless a other purity annotation is used.

Figure 7.2: Valid combinations of annotations on Methods.

Annotation	Fresh	FreshIf	NonFresh	Pure	Local	Ignore
Fresh	X				X	X
FreshIf		X		X	X	X
NonFresh			X		X	X
Pure		X		X		
Local	X	X	X		X	X
Ignore	X	X	X		X	X

A constructor permits the same annotations but with slightly modification. Local is implicit for constructors since they create the new state and assigns the default value to each field and doesn't have to be written out. Figure 7.2 marks all the valid annotation combinations for constructors.

Figure 7.3: Valid combinations of annotations on Constructors.

Annotation	FreshIf	NonFresh	Pure	Ignore
FreshIf	X		X	X
NonFresh		X		X
Pure			X	
Ignore	X	X		X

Parameters permit a single freshness annotation or Local. The annotation state what type of argument can be passed to the method. Ignore applies to the behaviour for the usage of the variable inside the method. In figure 7.3 the valid combinations are shown.

Fields allows the same annotations as Parameters with extension of Secret and the removal of FreshIf as shown in figure 7.4 . A local variable has the same rules as fields but disallowing Secret. There is only one annotation for types so there are no inconsistencies possible for types.

Then the type checker control that the class hierarchy is consistently annotated. This is done by controlling that each implementing method or overriding method has an equally or more restrictive annotation than the overridden method. No difference between sub

Figure 7.4: Valid combinations of annotations on Parameters.

Annotation	Fresh	FreshIf	NonFresh	Local	Ignore
Fresh	X			X	X
FreshIf		X			X
NonFresh			X		X
Local	X			X	X
Ignore	X	X	X	X	X

Figure 7.5: Valid combinations of annotations on Field.

Annotation	Fresh	Secret	NonFresh	Local	Ignore
Fresh	X			X	
Secret		X			
NonFresh			X		
Local	X			X	
Ignore					X

classing and interface implementation. The implementation must match the strictest definition.

The purity annotation is ordered as "@Ignore<unannotated<@Local<@Pure" where "<" denotes which is more restricted. This means @Local annotated method may override or implement @Ignore annotated and unannotated but not methods annotated @Pure.

The freshness annotation is almost independent of the purity annotations and for the class hierarchy to be consistently annotated an implementing or extending method must also have a freshness annotation that is consistent with the original definition. An ordering exists such that "@Ignore<unannotated<@FreshIf"<(@NonFresh/@Fresh)". The primary check preformed are listed in figure 7.6.

If a program does not violate these type checks then it has a potentially consistent set of annotations. The only remaining issue is that the methods actually has the correct behaviour restricted by the annotations. These rules guarantee that it is correct to use the statically known type of the caller of a method when determining the purity of the invoked method. The statically known super type method declarations gives the minimal purity and freshness condition of any method which overrides or implement that method. The restrictions only get stricter further down the type hierarchy. A circular type hierarchy which is one where loops of the kind “A extends B” and “B extends A” is of course only consistent if all effected methods have the same annotations. Otherwise at least one class will complain that they have lesser restriction than the class which it extends. The checks only need to check one step up the hierarchy.

Figure 7.6: Typing rules for the class hierarchy.

- A method may be annotated only equal or more restricted than any overridden method according to the listed ordering. This both according to the freshness and purity orderings.
- A method may be annotated only equal or more restricted than the interface definition of the method. This both according to the freshness and purity orderings.
- A method parameter may be annotated only equal or more restricted than its definition in an overridden method according to the listed ordering. This only according to the freshness ordering.
- A method parameter may be annotated only equal or more restricted than its definition in the interface specification of the method according to the listed ordering. This only according to the freshness ordering.
- In the case that a method implements both an Interface method and extends a superclass method the strictest annotations apply. This applies to both parameters and methods.
- A class with any method with stricter definitions in superclass than the same method signature in an interface cannot implement the Interface.

Example 35: Example demonstrating the type rules.

```
import lang.ast.SideEffect.*;
public class ExtendandImpl03 extends
  Parent implements Interface01{

  //Wrong missing @Fresh!
  public Integer freshMethod(){return 1;}
  //Wrong needed @Pure!
  @Local public Integer pureMethod(){return 1;}
}

public class Parent{
  @Pure public Integer freshMethod(){return 1;}
  @Pure public Integer pureMethod(){return 1;}
}

public interface Interface01{
  @Fresh public int freshMethod();
  public int pureMethod();
}
```

Method Checking

The checking is mostly modular, handling each method in isolation. The assignment statements and method calls are checked to verify if they are correctly used. For each method call there are two parts to the checking.

The checking needs to make sure that the passed arguments matches the appropriate target methods annotations. This is a verification of each expression provided as an argument if it has the correct freshness given the target methods parameter annotations. The expressions freshness can either directly be determined based on types and annotations on involved methods, fields and variables or determined via dataflow analysis. In the section 7.2 (Freshness analysis) I describe more in detail how the freshness analysis is done.

First the called methods must be determined to know which annotations should be followed. This means determining an exact type or a super type for the callers. The annotation rules assure that the analysis is safe as long as the type is a super type of all the possible callers but I want to have an as close as possible super type since stricter restrictions might apply to implementations for types further down the in hierarchy. That would be more information for the type checker. The less precise the type is the more false positive are possible when the super class methods are more precise.

The type is obtained using a simple data flow analysis which collects only the most constrained super type for each variable if the expression is a local variable or a parameter otherwise the statical declared type is used. The effectively final property and final annotation is used to aid the type deduction in which case the initiation or possible initiations are checked to determine the type. No attempt is currently made to limit the type of fields.

The example 36 exemplifies a case where more accurate type information effects the allowed calls. The statement "x.getOpen()" is okay if the type of x is resolved to "B" but not if its "A "as according to the declaration.

I construct a simple call graph in my checker to be used for two things. Firstly is for limited interference purposes. l determine what type of callers the method has to determine if it is appropriate to assume a freshness annotation or FreshIf annotation. If the checker determine that all calls are known to be from a constructor then local is acceptable. If all calls are from an NTA calculation then using the Fresh should be attempted despite having Pure already possible form the annotator.This is needed to allow combining NTA to use helper attributes.

The call graph operates under a closed world assumption.

The call graph is currently constructed using a static class hierarchy analysis (CHA) approach. A later change to a Rapid type based approach was planned to increase accuracy. CHA tries to connects a method call with all possible implementations that could be invoked which is the type of the caller and all suptypes [34]. CHA might however give false positive due to the overly conservative determination of the type of the caller of the method. According to [34] a change to rapid type algorithm would not be too costly but give greatly increased accuracy. A Rapid type algorithm determine a set of candidate types based on those that actually instantiated in the program or the method

Example 36: Example demonstrating the type rules.

```
import lang.ast.SideEffect.*;
public class Main{
        @Pure public static void main(String[] args){
                A p=new A();
                A x=new B();
                p.getOpen(); //Wrong!!
                x.getOpen(); // (Type dataflow = Okey, declared type= Wrong)
        }
}

public class A{
    public int getOpen(){
        return 5;}
}

public class B extends A{
        @Pure public int getOpen(){
                return 5;}
}
```

as far as can be determined.

The call graph uses the declared type at each method call and is constructed by storing the pairs of call sites and destination. The representation is a map over methods and caller sites in order accommodate this. Only calls sites either from an annotated method or which calls an annotated method are represented in the call graph. This is because the checker is only concerned about how protected values from attributes and annotated methods are used and called. When a method needs to obtain its callers a lookup is preform first of it self and then of the overridden or implemented method .

Freshness Analysis

The freshness of an expression determines if it may be change in any way. This concept is used to support that new objects may be modified before they are returned by a method. A method annotated local is only callable for a local annotated expression or a fresh expression.

The freshness information is firstly obtained for the expression type. The immutable types which in Java is String, Null and the primitive classes are always fresh due to being immutable. No work is needed when encountering these types. In other cases, the information needs to be deduced. This may involve both alias analysis and data flow analysis.

First the locality of the expression is traced to its top meaning access chain is reduced as much as possible or until known freshness. This is according to rules of the influence of *@Secret* and *@Local* annotations. These annotation influences the determination of the next access. For example, new Z().x.y might be of same freshness as new Z().x if

field "y" is Local. If this simplification for example encounters an object creation or array creation or a "fresh" method call then the expressions freshness is clear.

When the locality top is a variable the cases are that it is unqualified or qualified. In the unqualified case, then the annotations on the variable are controlled to find the top of the locality. Depending on the annotation on the field, method or variable which represent the top of the locality it determined if the freshness status is known or not. Annotations such as @Fresh, @Ignore, @Secret give information about the freshness status. If no quick conclusion can be drawn the checker starts to search backward in the Control Flow Graph (CFG) to find the current definitions of the locality and check their freshness status.

In the Freshness analysis, I utilizes a k-limiting approach to represented my access chains to be precise. I default to using 3 levels of access and a marker for a forth layer only indicating if in locality or not. The implementation is that first a map of all access chains used in the method is constructed using k-limiting. This creates one unique object for every access chain so results can be cached for the same access chain. The longest access chains are of length 3 and longer will be truncated. The chain “x.y.z.d.f” will be truncated to "x.y.z" if in locality or "x.y.z.NonLocality".

For any access chain which ends in the NonLocality marker no CFG traversing is necessary since these locations are not explicitly tracked and not influenced by any assignment to a tracked location. These untracked locations cannot be assumed to be fresh under any circumstances.

This k-limiting approach also reflect to my assumption about ASTs should be relatively immutable and there should not really be any need to change the node through complicated access chains. 3 or 4 levels of field sensitivity should be more than sufficient for typical changes. Another detail is that I use a single field representation of arrays which might be quite restrictive since the properties must apply to all locations for properties like freshness and nonnull. If a single assignment to an array is violating fresh then no locations can be assumed to be fresh or local.

In example 36 I try to illustrate how k-limiting works. I trunkate long access chains to avoid wasting computing power on representing all the extra field information that would be required to track that information both for aliases and the dataflow itself. I can’t think of a practical situation where one needs to manipulate new created object via long access chains. The while loop in the example represent the more reasonable that if we want object "x.x.x.x.x" to be fresh then we want every object accessed via "x" to be fresh and *@Local* is well suited to enforce that.

Example 37: Examples about k-limiting and locality.

```
public class Test{
   int x;
   Test t;
   @Local Test Lt;

   @Local public Test FreshnessDemo1(){
                Test tester = new Test(); // test1 = [Fresh]
                tester.t= new Test(); // tester.t = [Fresh]
                tester.t.Lt.Lt=tester;
                // The locality top for tester.t.Lt.Lt is tester.t
                //tester.t is deduced to fresh of
                //"new Test()" and assign valid.
                tester.FreshnessDemo1().Lt.Lt = tester;
                // Locality top is tester.FreshnessDemo1() which
                // has not Fresh assignment is thus wrong!!
                tester.t.Lt.t = new Test();
                Lt = tester.t.Lt.t.Lt;
                // The locality top "tester.t.Lt.t" is a to long access
                // chain. It unlikely that it should be fresh and expensive
                // to keep track of field information so deep in objects.

                tester.Lt.t = tester;
                Lt = tester.Lt.t.Lt.Lt
                // Locality Top = tester.Lt.t which is Fresh okay

                // Loop updating tester.i , ... , tester.t.t.t.i .
                // .Verifing all individually would be expensive way
                // of finding the the error at tester.t.t.t.i
                // especially if the loop is long.
                tester.t = new tester();
                tester.t.t = new tester();
                for (int i=0;i<4,i++){
                        tester.i=0;
                        tester = tester.t;
                }

                // @Local provides an infinite chain of
                // locations that are either null or fresh.
                  while(tester!=null){
                           tester.i=0;
                           tester = tester.Lt;
                         }
                return g()>3 ? t : tester;
   }


}
```

In the analysis, I search backwards in the CFG to determine if an access chain represents a fresh object or not. The analysis is implemented as a combination of a few attributes. A freshness attribute which determines the freshness for a given Java construction. This attribute invokes a helper attribute heapStatus(AccessChain a) when needed information from the CFG.

In the heapStatus(AccessChain a) attribute current definitions for the access chain across different control flow paths are evaluated to determine the minimum freshness of the access chain. The definitions evaluated in the heapStatus(AccessChain a) are obtained in a circular attribute I called currentDefinitions.

Example 38: Examples about k-limiting and locality.

```
// Simplified overview of Freshness analysis
// Freshness=local information if knownlocally otherwise
// Freshness=heapStatus(Access chain) (cfg lookup)
// heapStatus(Access chain)=
//              min(currentDefinitions((Access chain).Freshness()),

// Ignore>Fresh>Local>NonFresh>Maybe
public class Test {
        @Local Test LT;
        Test T;
        int x;
        @Fresh public test FreshAnalysis(int a){
                Test test = new Test(); //*1 // Freshness=Fresh
                test.T = new Test(); //*2 // Freshness=Fresh
                test.LT.T = new test(); //*3
                if (x>a){
                test.T =  FreshAnalysis(4); //*4 // Freshness=Fresh
                }else{
                test.LT.T = T; } //*5
                // Freshness=NonFresh
                if (x==10)
                test.T=test.LT.T; //*6
                // Freshness=heapStatus(test.LT.T)  ->
                //                       currentDefinitions={*2,*4,*5}
                test.LT.LT.T = test(); //*7
                return test.T.LT.LT;
                // Freshness=heapStatus(test.T)->
                //      min(currentDefinitions(test.T).Freshness()),
                // with currentDefinitions(test.T) = {*6,*4,*3}
                // Freshness=min(heapStatus(test.LT.T),Fresh,Fresh)
                // Freshness=min(min(Fresh,Maybe,NonFresh),Fresh)
                // Freshness=Maybe -> wrong because *4
        }
}
```

The implementation of currentDefintions is a critical point in the analysis the options where there were many options and I made several versions. One alternative was forward data flow analysis either flow insensitivity or flow sensitive and another was a backwards

collection of definition as they come. The second alternative is the one I use currently. The current implementation collects the definitions for every access chain requested up to where the access chain was requested.

Aliasing is not a problem for local fields since the rules for local field is that they hold fresh or local objects at all times. This is invariant of aliasing. Any aliased variable would simply be under the same restriction or harder restrictions since the annotations is on the field. An objects freshness is the freshness of itself and all it fields annotated *@Local*. These fields can only be assigned fresh objects in any method without visible side effects. Therefore, side effect free methods can't destroy freshness no matter aliasing.

Aliasing is only a potential factor for local variables and for fields without any *@Local* annotation.

Example 39: Aliasing example.

```
public class Test{
   @Local Test Lt;
   Test t;
   int i;

   @Pure public void AliasDemo(){
                Test tester = new Test();
                test1.t= new Test();
                Test aliaser = tester;
                aliaser.t = t; // tester.t no longer fresh
                tester.t.i++; //Wrong !
                // The same is not possible for Local field due to Locals rules

                aliaser.Lt = t; // Not allowed due to t not fresh.
                // Any aliasing variable can only place fresh objects
                // in which case the freshness is preserved
                aliaser.Lt = new Test();
                // tester.Lt consequently always fresh invariant of any aliasing;
                tester.Lt.t = t; // always acceptable
   }
}
```

7.2.1 Checking @Pure

The *@Pure* is the standard annotation for side effect free method.

To check that the method satisfy *@Pure* each so annotated method is traversed. Warnings might be triggered for reading a method, assigning a variable, calling a method and passing a variable as an argument to a method.

In the rules listed in figure 7.2.1, different type of statement is illustrated.

A problem in checking that the method only does the following is the aliasing problem that a local variable may alias a pre-existing object and an assignment to a field may not point to a newly allocated object. Thus, an limited alias analysis is also implemented.

Figure 7.7: Pure behaviours.

1. Function calls are allowed to methods annotated @Ignore, @Pure and @Fresh without any criteria.
2. Function calls are allowed to methods annotated @Local if called on a locally newly created object (Fresh). For example x.y.m() is valid only if x.y is "Fresh" according to the freshness analysis.
3. Function calls to methods annotated @Local may get passed Fresh objects to its @Local annotated parameters and any object to any non annotated fields.
4. A Pure method are not allowed to modify the referent of it parameters unless the parameter is annotated @Fresh.
5. A field annotated @Secret may only be read and assigned to if belonging to the same group.
6. A local variable may be freely assigned.
7. A referent may only be assigned to if it's a locally created object.
8. Any field without @Secret may be read.
9. The function may throw exceptions but not read them or manipulate them.

A second problem is if the function call is to unannotated methods it will be with all legacy code. Then the checker simply has to assume that they break rules. If the source is available inference can be done but my ExtJPureChecker does only attempt to do inference in a few special cases. For library method whose source can't be analysed an error message will be generated if not in a list of pure library methods provided.

The problem possed by unannotated methods has to be addressed with manual work and the list generating functionality of the checker. If the checker is asked to generate a list over all the methods that needs to be annotated. Then a guess based on current warning information is provided.

7.2.2 Checking @Fresh

For *@Fresh* on a method there are a few additional checks beyond those for *@Pure*. The extra tests regards the return value and the method type. In the figure 7.2.2 a few point about what must be checked for *@Fresh*.

The problem here is to determine that all return statements value is a fresh object.

Figure 7.8: Fresh behaviours.

- Allowed to do the same as @Pure.
- Must return an object created during it execution either internally or obtained via method call.
- All return objects should be fresh. A fresh object may be null.
- The type should be of reference type. A primitive numerical type is not allowed. Primitives are always fresh values since they are not objects so an annotation is redundant.
- If called by an assignment to a local field of an entity type then it should be non null. The same applies if the caller is a NTA then we want nonnull as value.

Checking @Ignore

The *@Ignore* annotation on a method causes the checker to skip the inside of the method and only check the method signature. As far as the warning generation is concerned a *@Ignore* method has no statements and consequently no warnings will be generated for any side effects and any call will be okay.

Since the method signature is still checked any other freshness and purity annotation on method and on the parameters still apply and causes warnings.

Furthermore, an Ignore method is not modelled by the call graph. The programmer has total responsibility to guarantee that such a method doesn't do anything unintentional.

When placed on any variable or parameter the @Ignore annotation guaranties that the variable can't trigger any warnings for any of its usage.

7.2.3 Checking @FreshIf

The *@FreshIf* annotation provides the polymorphism for the return value type. The annotation is important for the freshness analysis but has no specific checks. There is no warning apart from the inconsistent annotation warnings which is generated because the *@FreshIf* annotation. The freshness analysis will use this annotation when determining if a method returns a fresh object or not. For all parameters with this annotation and for the caller if the method is annotated the freshness analysis is invoked and if the determination is not Fresh or Ignore for each of them then the returned object is not fresh. This information can then trigger other warnings.

In example 40 a minimal example for *@FreshIf* is shown. In the example some complicated element selection code has been factored out from selected() and the returned object might or might not be fresh. The freshness depends on the objects given to a,b and link.

Example 40: *@FreshIf* example.

```
public class X{
   Int k;
   X link;
   @Local public X obtainselected(){
           X a= new X(); a.k=10;
           X b= new X(); b.k=3;
           link = new X();
           return a.selectX(link).selectX(b);
   }

   @FreshIf public X selectX(@FreshIf X a){
           // complex selection process!!!
           return a.k<this.k ? a : this;
   }
}
```

7.2.4 Checking @Secret

The @Secret annotation is the easiest since it a field only annotation. The check only involve study every field access and determining if it's in a method with the required annotation. The group should be correct.

Furthermore, to isolate secret states it's not allowed to read or write through a chain of Secret annotated fields. This is due to that secret field are intended to be object secret and only assigned via the corresponding object. A warning is thus issued when an access chain contains two or more fields annotated secret.

7.2.5 Checking @Local

The @Local annotation on the other hand requires more work to verify. It can be placed on both methods, field and parameters. When placed on a method it allows everything a

Pure method can do with the addition of assigning the current objects fields so that no longer triggers warnings. For an annotated parameter, the local changes to the parameter is allowed meaning warnings for these changes are silenced.

Figure 7.9: Local behaviours.

- Allowed to do the same as @Pure.
- May assign fields Annotated @Local with fresh objects in the corresponding object.
- May modify fields not annotated by @Local in the corresponding object.
- For an array annotated @Local the array may be modified but not the elements.

7.2.6 Checking @Entity

The entity annotation is placed on the classes for AST nodes, to denote that they are entity objects, rather than objects that represent values. There are a few specific checks. When an entity type is passed to as a fresh parameter or to a field which demands a fresh object then the analysis tries to determine that it's also non null. Null is otherwise allowed in those contexts. This is to meet the requirements that nodes in the children of an AST node are always complete sub trees and never refer to null. The correct way for entities is to use an object as the null value.

On the other hand *@Entity* modify the behaviour for assignment to fields which doesn't want a fresh object. They are not allowed to be fresh any more. In way normal reference attributes are not allowed to reference to new ASTNodes but if the type is not an entity then it may be fresh. JastAdd "Tokens" can escape this limitation with exception for token fields which may be assigned any value.

8 Applying the annotations for JastAdd

There are several different forms of impurity that we would want to detect and warn against regarding the code generated by JastAdd. I presented two introductory examples in the section 3 as motivating examples. In this section I will go through all the JastAdd semantic constructions and address how each work with the checker and how they should be annotated. The different constructs used in a specification to JastAdd which I have considered are

- **Type declarations**: Programmer specified class and Interfaces declarations doesn't need any annotations. The *@entity* should be provided by the programmer if needed. For JastAdd only the implicit AST root class ASTNode need *@entity*.

- **Methods and Constructor**: Java method and constructors specified by the programmer need to be annotated depending on usage. The programmer can specify the annotation separately in annotation files as an alternative to modifying source code. Many implicit methods need annotations. AST constructing gets *@Local* or *@Ignore* depending on if they are supposed to be exposed to the programmer.

- **Primitive Attributes**: There are two primary types of attributes. They are either synthesised or inherited. An attribute is converted into mainly two methods which is a retrieving method and one calculating method for each equation. The getter can manipulate secret state and the calculation can not. These are annotated *@Pure*. Any cache field is annotated *@Secret* to protected the field.

- **Parametrised Attributes**: Attributes may be given parameters. They are cached in map instead of directly to a field which forced the analysis to allow this modifications to a *@Secret* protected field. They are otherwise treated in the same way as primitive attributes.

- **Collections Attributes**: Collection attributes generates more methods for their implementation. In addition to the normal methods there is a new contribution method for potential addition which performs a local change to the collection passed as a parameter. A survey method performs a similar construction of the set of all nodes which might affect the collection.

- **Non Terminal Attributes**: NTAs requires a fresh object from their computation method but the getter method should still be *Pure*. This approach works well until we encounter NTAs which enlists helper attributes for the calculation. The helper attributes need to be assumed *@Fresh* by the call graph heuristic if they are only referenced by the NTA.

- **Circular Attributes**: Circular attributes is attribute which needs to be evaluated to fix point. This is no new problem since every attribute is only modified in its own equation and then either form a completely fresh value or from its own previous

value. The checker has no problem allowing this with the same annotations as for non circular attributes under the rules explained in the previous chapter.

- **refines**: A refines replaces the definition for another synthesised attribute and are given no special treatment by the checker. I haven't considered them much. Unless they reference the original definition they completely replaces it and the checker only sees the new attribute otherwise the redefinition is one attribute that simply references a compute method. The computation method used the same annotation as the original.

- **rewrites**: Rewrites are constructs which replaces one node with a new one in. They utilize the same structure as NTAs except that the computation method is divided into many smaller rewrite rule methods for each scenario in which the node may be replaced. These are also annotated *@Fresh*.

The Java that needs to be checked is described. For some constructs, complications can arise when combined with other code and for those I explain how the checker can deal with them.

The examples I present here are described on a AST generated for a small abstract grammar shown in example 41. I also used the examples from JastAdd test suits used for JastAdd development [61].

Example 41: small test grammar as context for the examples

```
//A simple test grammer for the testing
Tree :: A* B; // A top class that has a list of A:s;
A ::=<ID> C* B; // Independent class 1
B::=<ID>; // Independent class 2:
C ::= Cond:Expr<ID>; // parent to D,E.
D : C ::= Elist:Expr* <ID>;
E : C ::= <ID>;

abstract Expr;
Numeral : Expr ::= <NUMERAL>;
```

8.1 Abstract Grammar and Internal methods

When working with JastAdd an AST class structure is built by from an abstract grammar. This grammar is used to generate the AST classes with the fields needed to hold the children and values of the nodes. Children to an AST class are an AST class, an instance of opt or list or a NTA class. NTA classes have no implementation generated by the JastAdd system but must be defined by the programmer. The generated code should be considered a pure implementation and generate no warnings to the programmer.

The default generated code will however contain setters and getter for internal fields used by JastAdd. JastAdd automatically generates ASTNode, List and Opt as default classes. For the default AST classes fields storing the parent and children are assign be such default code. Apart for code for constructing the AST there several other methods are provided. JastAdd provides an iterator over children and methods for constructing deep and shallow copies of the AST. Much of the code is likely to be invoked by use in attributes and rewrites and must be annotated.

Example of this is are the “init$children”,”addChild(T node), setChild(ASTNode node,int i)” methods which should be allowed to be used only when it’s allowed to change the object. That would apply in constructors but also for addition of a NTA and during rewrites. Warnings should otherwise be issued for any attribute calculation or @Pure method that tries to use these methods.

Example 42: Internal JastAdd methods and fields

```
package test;

public class A{
        // Constructing AST
        @SideEffect.Secret(group="_ASTNode") private int childIndex = -1;
        @SideEffect.Secret(group="_ASTNode") protected ASTNode parent;
        // Local to control rewrites authority and modifiablity
        @SideEffect.Local protected ASTNode[] children;
        @SideEffect.Ignore public void init$Children() {
                children = new ASTNode[1];
                setChild(new List(), 0);
        }
        // Force all childs to be newly created nodes to assure
        @SideEffect.Local public void addChild(@Fresh T node) {
                setChild(node, getNumChildNoTransform());}
        @SideEffect.Ignore public void setChild(@Fresh ASTNode node, int i) {
                . . .}
}
```

Several methods such as "clone()", "treeCopy()", "getParent()", "numChildren()" should be callable by the programmer without generating any warnings. All these methods need to be annotated with an appropriate annotation to allow both the default generated code to type check and for the methods to be useable in user code.

8.2 Methods and Equations

There is no major distinction between methods and equations and attributes for the purity checking once the Java code has been generated. They all results in methods been generated.

The JastAdd system should generate annotations for each method corresponding to an equation or the appropriated list of functions that should be pure.

For use with the purity checker JastAdd would need to generate annotation to specify that methods associated with an equation needed annotation for purity checker. An equation or method involved in constructing a NTA needs to create a new AST node apart from simple being pure.

The cache resetting methods are special and needs to be annotated *@Ignore* since they must be allowed to reset the cache fields which are protected by *@Secret*. In example 43 the top level cache resetting methods are shown. A method for resetting all attribute and the smaller resetting methods for types of attribute and finally individual attributes.

Example 43: Cache resetting example

```
package test;

public class A{
        @SideEffect.Ignore public void flushAttrCache() {
                super.flushAttrCache();
                attr_String_reset();
        }
        @SideEffect.Ignore public void flushCollectionCache() {
                super.flushCollectionCache();
                A_b_visited = false;
                A_b_computed = false;
                A_b_value = null;
                contributorMap_Node_b = null;
                . . .
        }
}
```

8.3 Primitive Attributes

An attribute works as virtual methods added to classes. An attribute can either be inherited or synthesized meaning defined in the context of a parent in AST or in the object itself. This is what separates them from normal methods.

An attribute is declared by an equation which is a block of Java code that needs to be pure. Every attribute declared in aspects must be checked so that it's the Java code in the equation is pure.

An attribute is translated by the JastAdd system, in all but the case when the attribute is defined by single expression, into at least two methods. One that does the calculation ("calculate-methods") and one that preform caching and circularity checks and retrieves the value ("retrieve-method"). This structure is common for all types of attributes including the collections and NTAs. The differences lie in which additional methods are needed. In the case of Inherited attributes there is "lookup methods".

Example 44: Synthesised Attribute

```
Aspect Pure{

syn String A.synAttr(){
// something pure!
}

}
```

Both the compute and retrieve methods are annotated *@Pure*. The checker then complains about any side effects in the compute method which contains the code given definition to the attribute.

Example 45: Inherited Attribute

```
aspect Pure {

inh String B.Amessage();

eq A.getB().Amessage(){
//Must be pure !
return "Parent:A";
}
eq Tree.getB().Amessage(){
return "Parent:Tree";
}

}
```

In the case of an inherited attribute the value is defined in other objects. A lookup method is used to queries a parent in the AST for the value and that parents class would have either a method for searching further up the AST or a calculate method to obtain the value of the attribute. All involved lookup, retrieve and calculate methods is

annotated *@Pure*. This assures side effect freeness of the attribute and correct behaviour when used by other attributes and methods.

Example 46: Generated code for primitive attributes

```
public class A{
    /** @apilevel internal */
  @Secret(group="_ASTNode") protected boolean synAttr_computed = false;
  @Secret(group="_ASTNode") protected boolean synAttr_visited = false;
  @Secret(group="_ASTNode") protected String synAttr_value;

  @Pure private String synAttr_compute() {
   // somthing pure
  }

  @SideEffect.Ignore private void synAttr_String_reset() {
  synAttr_value = null;
  synAttr_visited = false;
  }

  @Pure(group="_ASTNode") public Object ASTNode.synAttr(){
   . . .
  if (synAttr_computed) {
    return synAttr_value;
  }
   . . .
  synAttr_value = synAttr_compute();
  synAttr_computed = true;
  synAttr_visited = false;
  return synAttr_value;
  }
}
```

Parametrised Attributes

Parametrised attributes both inherited and synthesized work the same way as the unparametrized versions as far as the created methods and their desired annotation goes. The difficulty is in the caching where the attributes are placed and retrieved from inside a map. This forces that apply methods to a secret field variable must be allowed.

8.4 Collection Attributes

Collection Attribute is a type of attribute that are built from many contributions collected in some class. The attribute is declared to belong to some class and then one or several AST classes can contribute values to the collection. All the contributions are when requested gathered by the JastAdd system that traverse the AST invoking contribution methods for everyone.

Example 47: Generated code for a parametrised attribute.

```
package test;

public class A{
  @Secret(group="_ASTNode")
  protected java.util.Set synAttr_String_visited;
  @Secret(group="_ASTNode")
  protected java.util.Map synAttr_String_values;

  @Ignore private void synAttr_String_reset() {
    synAttr_String_values = null;
    synAttr_String_visited = null;}

  @ASTNodeAnnotation.Attribute(kind=ASTNodeAnnotation.Kind.SYN)
  @ASTNodeAnnotation.Source(aspect="Test", declaredAt="x.jrag:2")
  @Pure(group="_ASTNode") public boolean synAttr(String s) {
  Object _parameters = s;
  . . .
  if (synAttr_String_values == null)
    synAttr_String_values = new java.util.HashMap(4);
  . . .
  if (synAttr_String_values.containsKey(_parameters))
    return (Boolean) synAttr_String_values.get(_parameters);
  . . .
  boolean synAttr_String_value = s.equals("");
    synAttr_String_values.put(_parameters, synAttr_String_value);
    . . .
    return synAttr_String_value;
  }

  @Pure public boolean synAttr_compute(String s) {
    // Something pure
    return true;}
}
```

Example 48: Collection Attribute Example.

```
coll ArrayList<String> A.allchildren()
[new ArrayList<String>()] with add root A;

syn boolean B.condition()=true;

B contributes getID()
  when condition()
  to A.allchildren();
C contributes getID()
  when true()
  to A.allchildren()
  for relevantAs();
```

The important things for the correct behaviour of a collection attribute is that the method declared as contribute method only makes a modification to the collection method e.i satisfy @Local. The contribute method is declared after the keyword "with". In the example "add" is the contribute method.

The contributions them self needs to be side effect free in their construction and so also the condition on whether a contribution should be made. This means the Java expressions after the keywords "contributes","when", "to", "for" needs to be side effect free. They will all be represented in the generated code in pure methods. The traversing methods also needs to be pure.

The side effect check of the initial value expression will be performed since this expression is executed in the "retrieving" method. The problem is that it should also be fresh.

In the example 49 code for an synthesised collection attribute is shown. The user provides an initial value which must be a fresh value. The first part of the computation is to collected contributing nodes. This is done with a surveying method which in the example is "survey_A_coll". It the uses the "collect_contributors_A_coll(this,contributorMap_A_coll)" methods which adds contributing nodes to a map of contributions. The map representation allows the system to keep track over many different collections for with the same "root". A root is a node which serves as the starting point for the collection with contributions collected form the subtree with that root. In order to filling the Maps and annotations the shown annotations are needed.

Example 49: Generated code for a collection attribute.

```
public class A{
  @ASTNodeAnnotation.Attribute(kind=ASTNodeAnnotation.Kind.COLL)
  @ASTNodeAnnotation.Source(aspect="Test", declaredAt="X.jrag:10")
  @Pure(group="coll") public ArrayList<String> coll() {
    ...
    A_coll_value = coll_compute();
  }

  @Fresh(group="coll") private ArrayList<String> coll_compute() {
    ASTNode node = this;
    while (node != null && !(node instanceof A)) {
      node = node.getParent();
    }
    A root = (A) node;
    root.survey_A_coll();
    ArrayList<String> _computedValue = new ArrayList<String>();
    if (root.contributorMap_A_coll.containsKey(this)) {
    for (ASTNode contributor : root.contributorMap_A_coll.get(this)) {
      contributor.contributeTo_A_coll(_computedValue);
      }
    }
    return _computedValue;
  }
  // Helper method
  @Pure(group="coll") protected void survey_A_coll() {
      if (contributorMap_A_coll == null) {
```

```
        contributorMap_A_coll = new
        java.util.IdentityHashMap<ASTNode, java.util.Set<ASTNode>>();
        collect_contributors_A_coll(this, contributorMap_A_coll);
      }
    }
    // Collect contribution
  @Pure protected void collect_contributers_A_coll(A _root, @Local
      HashMap<ASTNode, java.util.Set<ASTNode>> _map){
    if (condition()){
      A target = (A) (a());
      java.util.Set<ASTNode> contributors = _map.get(target);
      if (contributors == null) {
      contributors = new java.util.LinkedHashSet<ASTNode>();
      _map.put((ASTNode) target, contributors);
      }
    }
    super.collect_contributors_A_set(_root, _map);
  }
  @Secret(group="coll") protected java.util.Map<ASTNode,
  java.util.Set<ASTNode>> contributorMap_A_coll = null;
  @Pure protected void
   contributeTo_A_coll(@Local ArrayList<String> collection) {
      super.contributeTo_A_coll(collection);
      if (condition())
        collection.add(getID());
    }
}
```

8.5 Circular Attributes

Some attributes can be dependent on themselves. An example is dataflow calculations [9]. These are calculated to fixed point. For the side effect analysis, they however don't impose any new mayor problem. The calculation and retrieving method should both still be annotated @Pure the same as for other attribute types.

Example 50: Circular Attribute.

```
Aspect Purity {

syn Set<Function> Function.reachable() circular
[new HashSet<Function>()]
{
Set<Function> temp = new HashSet<Function>(collectFunctionCalls());

  for(Function f : collectFunctionCalls())
  {
    temp.addAll(f.reachable());

  }
  return temp;
}

}
```

In the example 51 code for an synthesised circular attribute is shown. A user provided initial value needs to be provided for which, in contrast to the initial value for collection attributes, no requirement is placed on freshness. Furthermore in order to deal with circularity more internal fields are necessary to keep track on the internal status as compared to none circular attribute. The new status fields are like all other fields part of JastAdds internal calculations annotatede *@Secret(group="_ASTNode")*.

Example 51: Generated code for a circular attribute.

```
public class A{
    /** @apilevel internal */
  @Secret(group="_ASTNode")
  protected boolean circAttr_computed = false;
  ------- protected boolean circAttr_visited = false;
  ------- protected String circAttr_value;
  ------- protected boolean circAttr_initialized = false;
  ------- protected ASTState.Cycle circAttr_cycle = null;

  @Pure private String synAttr_compute() {
   // somthing pure
  }
  @Pure private String initial() {
   // A inital value (user provided)
  }
  @SideEffect.Ignore private void synAttr_String_reset() {}
  @ASTNodeAnnotation.Attribute(kind=ASTNodeAnnotation.Kind.SYN,
      isCircular=true)
  @ASTNodeAnnotation.Source(aspect="Test", declaredAt="X.jrag:3")
  @Pure(group="_ASTNode") public String ASTNode.circAttr(){
    if (circAttr_computed) {
      return circAttr_value;
    }
    ASTState state = state();
    if (!circAttr_initialized) {
      circAttr_initialized = true;
      circAttr_value = inital(); //  bottom value
    }
    if (!state.inCircle() || state.calledByLazyAttribute()) {
      state.enterCircle();
      . . .
      circAttr_computed = true;
      . . .
      state.leaveCircle();
    } else if (circAttr_cycle != state.cycle()) {
      circAttr_cycle = state.cycle();
      if (state.lastCycle()) {
        circAttr_computed = true;
        boolean new_circAttr_value = synAttr_compute();
        return new_circAttr_value;
      }
      boolean new_circAttr_value = synAttr_compute();
      if (new_circAttr_value != circAttr_value) {
        state.setChangeInCycle();
      }
      circAttr_value = new_circAttr_value;
    }
    return circAttr_value;
  }
}
```

8.6 Non Terminal Attribute

NTAs represents new branches of the AST created via an equation. They must be newly created ASTNodes in order to be insertable into the AST having a single parent. The single unique parent is needed for the calculation of attributes traversing the AST such as inherited or collections to be calculatable.

Example 52: NTA Examples

```
Example 1
nta A ASTNode.getA(){
// has to produce a fresh object and be pure
 A node = new A(new B(ID), new B(ID));
return node;
}

//Example 2
nta A ASTNode.getA()=creation();
syn A ASTNode.creation()=new A(new B(ID), new B(ID));
```

NTA differ from normal attribute by demanding the creation of fresh object. Therefore, NTAs will have here computation methods annotated fresh.

Circular NTA

A NTA can be circular or using another attribute. In this case explained methodology of having unobservable computation methods fresh and the visible method pure wouldn't work since a NTA must receive a fresh object. An exception must be utilized in these cases. This uses the rule that an attribute that's only called from a NTA or more commonly from the associated computation method may be consisted to *@Fresh* instead of *@Pure* if its only called from a fresh method part of a NTA calculation.

In the example 53 code for a synthesised NTA is shown. The difference for NTAs compared to the other types of attributes is that the compute method is now annotated *@Fresh*. NTAs uses "setChild" to add the result as a new node in the AST which therefore needed to be annotated as I showed in example 42.

Example 53: Generated code NTA.

```
public class ASTNode{
  /** @apilevel internal */
  @SideEffect.Secret(group="_ASTNode")
  protected boolean getA_visited = false;
  @SideEffect.Secret(group="_ASTNode")
  protected boolean getA_computed = false;
  @SideEffect.Secret(group="_ASTNode")
  protected A getA_value;

  @ASTNodeAnnotation.Attribute(kind=ASTNodeAnnotation.Kind.SYN,
      isNTA=true)
  @ASTNodeAnnotation.Source(aspect="Test", declaredAt="X.jrag:2")
  @SideEffect.Pure(group="_ASTNode") public A getA() {
    . . .
    A getA_value = getA_compute();
    setChild(getA_value, getAChildPosition());
    . . .
    A node = (A) this.getChild(getAChildPosition());
    return node;
  }

  @SideEffect.Fresh private A getA_compute() {
    // Pure effects
    A node = new A(new B(ID), new B(ID));
    return node;
  }

}
```

8.7 rewrite and refine

Rewrite constructions can be used to replace a node in the AST with a new node of any node type either unconditionally or conditionally. These are conditions are called rewriting rules. Many different rewrite rules can be specified with different conditions and rewrites. The rules and the corresponding rewrites are the applied recursively in priority order as long as any condition hold.

The implication for purity analysis is that attribute values only need to be deterministic for a given AST. When an ASTNode has changed type the attribute may change. JastAdd will not cache any attribute until the AST is in its final rewritten form after all rewrite rules have applied until no longer applicable. Any client code accessing an attribute will however only see the final value since the rewrite rules will apply when an ASTNode is referenced from its parent.

Purity is important since the rewrite rules can apply in any order but it is required that the result should be the same AST independent of the order or how the AST is traversed. Otherwise the modelled language could behave unpredictable.

For unconditional rewrites the equation creating the replacing AST class must be annotated *@Fresh* and for the conditional rewrites the rewrite conditions that can depend on attributes also must be annotated *@Pure*. The purity checker will guarantee that the code doesn't contain any side effects. The situation during rewrite is special in that the node being rewritten is effectively fresh during the process. A parametrisation of *@Fresh* indicates that the current node should be treated as fresh during the method.

Example 54: Rewrite Example.

```
// rewrite calls a impuremethod or equation
rewrite B{
  when (ImpureCond()>3)
  to D{
  // Should generate a fresh object
  return new D();
  }
}

int B.conf = 3;

syn lazy int B.ImpureCond(){
int a=conf;
conf++;
return a;
}

//In this case impurity should be reported
for the condition method
```

There is something called circular rewrites which is rewrites depending indirectly on

the rewrite itself. These are implemented using circular NTAs and thus handled the same way.

In example 55 an example of the code generated for a rewrite is shown. A few methods to indicate that a node can be rewritten has been added as well as one rewriting method for each rewrite rule.

Example 55: Cache resetting example.

```
public class A{
  @SideEffect.Local public ASTNode rewriteTo() {
    if (RewriteCondition0()))
      return rewriteRule0();
    return super.rewriteTo();
  }
  // Rewrites should allow itself to be rewriteable
  @Fresh(rewrite=true) private D rewriteRule0() {
    // Create new C node
    D newD= new D(...);
    newD.children[1]=this.children[2];
    newD.children[0]=this.children[1];
    newD.setB(getB());
    return D;
  }
  /** @apilevel internal */
  @Pure public boolean canRewrite() {
    if (RewriteCondition0()) {
      return true;
    return false;
  }

  @Pure public boolean mayHaveRewrite() {
    return true;
  }

  @ASTNodeAnnotation.Attribute(kind=ASTNodeAnnotation.Kind.SYN,
      isCircular=true, isNTA=true)
  @ASTNodeAnnotation.Source(aspect="", declaredAt=":0")
  @SideEffect.Local(group="_ASTNode") public ASTNode rewrittenNode() {
    // ... circular NTA more or less ...
  }
  // Circular NTA fields ....
}
```

9 Evaluation

I tested the constructed purity checker and the JastAdd annotator in several ways. Since it uses similar annotations to JPure I could as a first check see that code annotated with JPure typecheck in ExtendJPure excluding @Entity specific warnings.

9.1 Running on Examples

During the development of the checker continuously as new feature or problems were discovered new test were constructed as testing for the tool.

Running JPure on examples generates correct code for the situations JPure can handle. Excluding my slight redefinition for arrays. This is by the design of my checker. My checker is more advanced than JPure and less conservative which means that when JPure determines a method to be of one purity level my checker will either agree with JPure, or determine that the method has even greater purity and can be annotated with more details. My purity type system is after JPure's and designed to be compatible with JPure for the situations Pearce considered in his paper [10].

The JPure examples was included in the test suit for the tool and is part of the basic testing of the checker. They cover a lot of Java situations but is in no way complete and the tests are all very simplistic. The latest publish version of JPure could not deal with all the tests in its own test suit when run with its own test script where some test were ignored. JPure can handled them using the include partly annotated Java1.5 runtime.

My checker can with its own suggested external annotations handle all the JPure suit test according to the Pearce' s manual annotations including the once.

I made several variations of the JPure test suit with small changes to check that the tool closely agreed with JPure and to check that lessening of annotations would induce errors. The result was that it does closely follow JPures annotations.

I wrote many new tests as I encountered problems and addressed them. My new test address questions regarding more complicated inheritance situation and control flows, anonymous class among other things.

A few tests from the JastAdd test suits[61] served as significant guide discovering problems and situations needed to be addressed since it contains many examples with all the different JastAdd constructions. JastAdd is the target application after all. The JastAdd examples contains lots of varied side effects to be mostly ignored but under various circumstances need to be checked in different ways. From the JastAdd tests I was forced to address questions of how unobservable state is allowed be used and modified specially when storing composite objects such as arrays and collections.

The JPure test suit consisted of 68 test cases where mine has around 200 test cases. My tests however are on average quite large with the entire JPure suit being counted as one test in my test suit.

9.2 Running on existing grammars

Testing of the complete solution was first done on minimal grammars with only one attribute to check. This was done to make sure that the annotations of the JastAdd annotator was reasonable enough to not trigger warnings for correct grammars.

The first real grammar the solution was tested on was my SimpliC compiler constructed for a compiler course. It parsed a very small subset of C consisting of basic method calls, numerical algebra and "if" conditionals along with some basic name and type analysis. In SimpliC I added an analysis which constructed the call graph for the program. It was designed very imperatively like I programmed normal Java and I didn't utilize collection attributes which could have simplified the code. In this code for this analysis I did overwrite parameters but otherwise all my attribute is side effect free. I didn't do code generation via attribute and it couldn't be easily converted since there I do have side effects. After allowing the parameter to be changeable my SimpliC program is without any side effects. After some work I determined that the checker result that SimpliC doesn't contain any side effects is a reasonable result. Any side effect containing code are well separated from any attribute calculation. The result of 0 warnings seems correct so I don't detect any mistakes there. SimpliC was a small enough compiler that I could keep it side effect free.

Next the purity checker was tested on a small lambda calculus compiler made by Niklas Fors. It posed as a test using NTAs and library code. The NTAs was only simple NTAs and not complicated circular NTAs. With only a few annotations in the user provided code or in external annotation file the example could be purity checked without errors. The example contained only one problem according to the checker a theoretical possibility for returning null as a NTA. This because the default method in the abstract class “Expr” return “null” and this then used for a NTA. NTAs can't be null according to the rules. My checker can't determine that the abstract method couldn't be invoked an issue a warning and It would be more appropriate to use a null object or throw an exception other than NullPointerException.

The example also helped me notice that I had missed to handle all Java statements. The compact conditional statements on which is the statement on the form "cond ? iftrue : iffalse" was not handled by my checker at that time. I had missed that for some reason. Thanks to Fors example I could address this and fix this gap and other gaps.

A try was made to test the tool on the compiler Fuji[21] but I can't have much confidence in the result since after generating the source code I cannot get the full program to compile without Java errors. This might be due to Fuji requiring some dependencies I miss and configured to be build using an old JastAdd version which might be incompatible with my annotating JastAdd version. Through some modification in the build scripts I have in any case obtained an AST. For the Fuji code I get many potential warnings. I reduced the number by identify error in how I handled rewrites which is used quite extensively in Fuji. Still quite a few remained but most of the errors are though problems with suggested annotation which caused some attribute construction code to be introduced. Removing all the noise few genuine side effects would remain. I know that there are side effects which have been detected. I have confirmed that Fuji uses manual

cache clearing apart for JastAdds own clearing method which is not how attribute should be cleared among other things.

ExtendJ on the other hand is no problem to compile. Significantly less warnings than for Fuji. A few problems were reported by the checker. From attribute code requests the parsing of additional Java source occurred. In which a counter counts the number of parsed Java sources and other side effects. The orgin for the error however is difficult to find with my checker currently not providing sufficient tracing information. The iterative addition of classes to the checking process currently doesn't save a complete trace over the information something that needs improvement. In any case sorting away the false errors is time consuming.

Also, there are some work with type constraint where information is collected and extracted in field in a static class. More warnings are generated due to the inaccuracy of the suggested annotations.It takes time to correct when you don't know which method does what. The warnings in example 56 shows a small subsection of the warnings generated.

Example 56: Example of some warnings in ExtendJ.

```
in ast\ConstantPool.java:ConstantPool
160:return labelCounter++; violates the annotations
at labelCounter
160:labelCounter:write effects the this object
but this object is not being constructed and
thus the method is impure

ast\ClassSource.java

141:program.numJavaFiles += 1; violates the
annotations at numJavaFiles 141:program.numJavaFiles:
write to visible field in an object that is not Fresh
because alias for parameter program that cannot be
assumed fresh on entry.

in ast\ClassSource.java:ClassSource


ast\ConstructorDecl.java
Called by org.extendj.ast.ConstructorDecl.createBCode(Unknown)
on line 871

197:int index = gen.constantPool().
addFieldref(classname, name, desc); violates the annotations.
The method call CodeGeneration.constantPool() called
for gen is local but the caller is not fresh or in
a modifiable locality.
The caller is not fresh because parameter gen cannot
be assumed fresh on entry but only Maybe.
```

```
ast\NumericLiteral.java
Called by org.extendj.ast.NumericLiteral.rewriteTo()
on line 570

584:return literal; of Freshness NonFresh cannot be
returned from a @Fresh method because literal =
parser.parse() initizises the variable to parser.parse().
 That is not Fresh but only NonFresh->
The method call org.extendj.ast.NumericLiteralParser.parse()
of puritylevel Local doesnot produce a new object.

in ast\NumericLiteral.java:org.extendj.ast.NumericLiteral
```

For *ExtJPureChecker*, itself there is the warnings for ExtendJ which is in the basis for the analysis. Additionally, there is warnings because how Jesper made the construction of the predecessors to the CFG. This construction is side effect full. In order to avoid this side effects all control flow graph constructor would have to be performed before the main analysis instead of during the analysis on demand. This is the added warnings because of the SimpleCFG. The checker itself otherwise doesn't introduce any new warnings after the annotations provided. I have some manual caching and resetting but these are annotated as to be valid or sufficiently seperated from the attribute code.

9.3 Performance

On the tested examples and JastAdd compliers the runtime for an iteration of the checking either for generating the library annotations or for generating warnings take only slightly more time than the compilation.

In the case of external library files having to be generated it could take a magnitude more time since annotations are generated in iterations with each iteration taking about the compile time of the program. Designing the generation to be done in iteration gives less data to be manually corrected in each iteration but make the initial generation of all the annotations information slow. If 6 different iteration cycles are required the then that will take 6 times the checking time. Fortunately, this is work that is only preformed once and then reused.

The runtimes for the checker on different programs is displayed in figure 9.3. I measure the *ExtJPureChecker* performance with the time it takes to generate and compile the code with JastAdd and Javac via the build scripts provided with the tools. I first clear all generated files and then I run the script and checks the time reported.

For my own checker I use an open source timer application since windows command prompt doesn't provide any. The timer times how long it takes to run the checker. The checkers two functions are timed in different groups. The time it takes on average to run an iteration of the annotator and how many iteration are needed. I measure with deciseconds accuracy.

The third statistic I measured is the suggest annotation feature were I measure the average time per annotation iteration. The biggest programs requiring upto 7 iterations

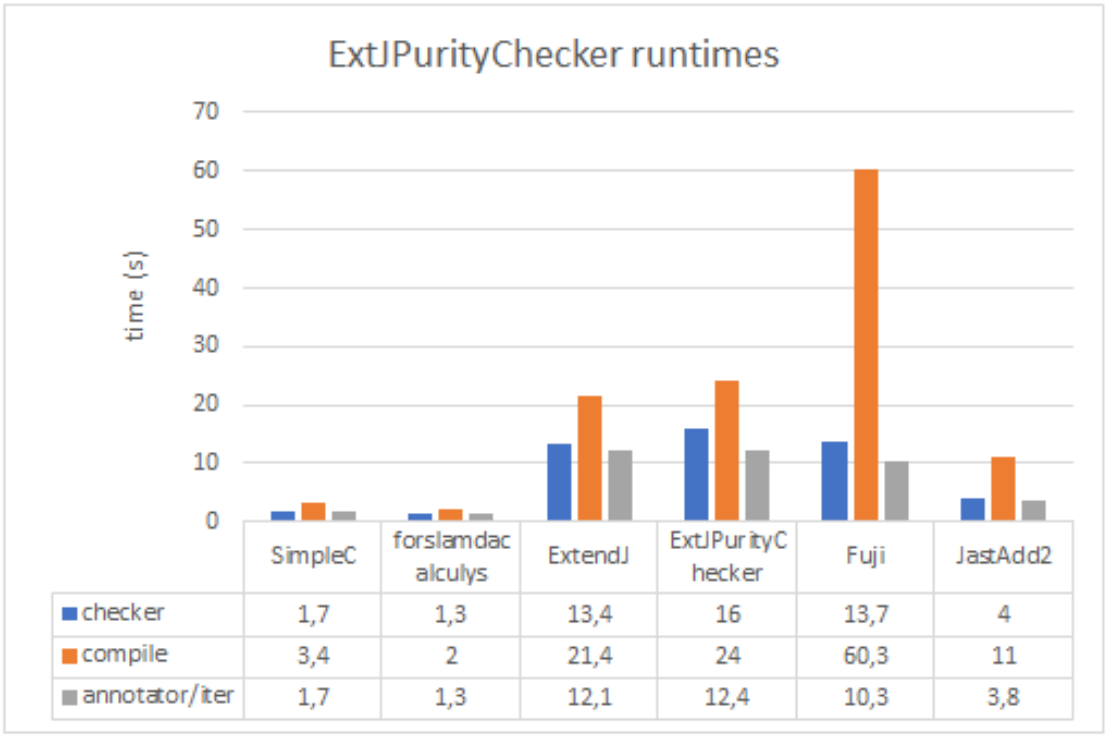


Figure 9.1: Purity Checker runtimes.

of loading. The number of needed annotation iterations are shown in figure 9.3.

The result is that the checker in its current implementation tend to be faster than the compilation once all annotations has been generated. This is show in figure 9.3 where the checker runs for 13,4s on ExtendJ and the compilation takes 21,4s. The reason for this is that most of the attributes tend to be defined by very small equations or methods mostly consisting of method calls. This can be evaluated quickly. Object manipulation via deep access chains and other modification which are time consuming to check are used fairly rarely. Then it might also be an indication that my approach could have been more precise but I think for very regular application of the tool that a doubling or less would be descent. Its for future work to allow the programmers more control and implement more precise tracking of structures and have a version that separates array accesses more. As ExtendJ becomes more supportive of Java 8 annotations , gets improved type inference for Java 8 and support more Java 8 features new options new opportunities would open up to expand this purity checker to handle different level of analysis and to address these features.

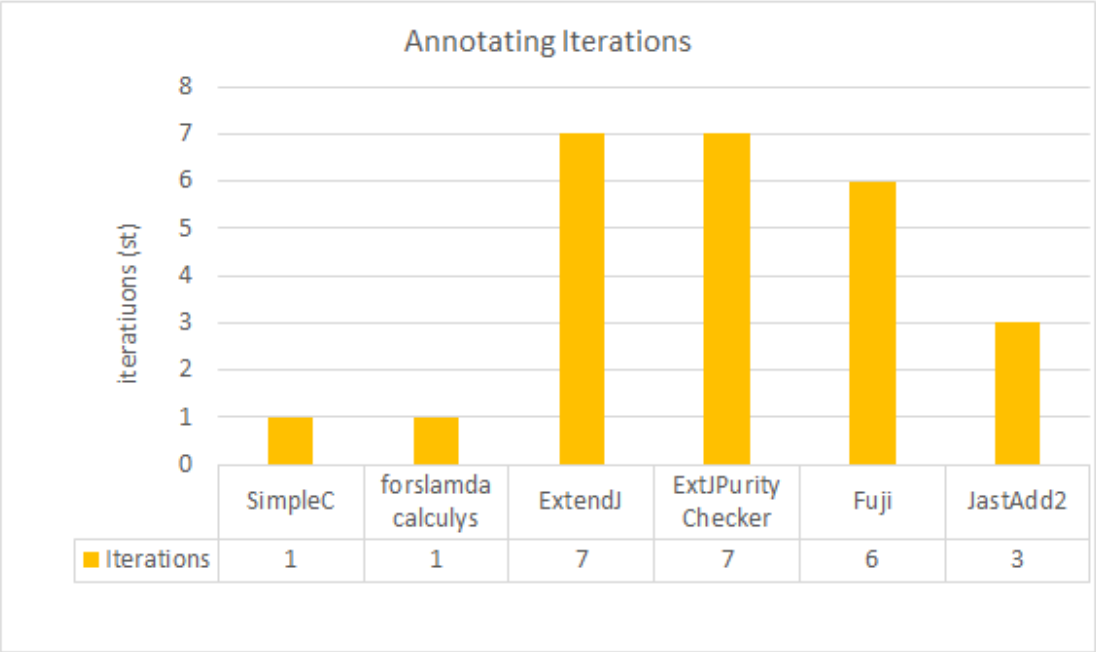


Figure 9.2: Number of annotation iterations.

10 Discussion

The development of the purity checker has had several setbacks and unexpected developments along the road.

When I decide to build my checker, I didn't look around much for which compiler. I was already biased towards using ExtendJ from initial discussions with my supervisor. I began working with ExtendJ without really consider the alternatives much. It was easy to work with ExtendJ due to access to the developers on the University, clear documentation and available visualisation AST tools. I didn't really consider ExtendJs limitations regarding bytecode manipulation.

A more capable checker might have been developed if Soot or ASM had been used from the begin due to byte code analysing and access to the pre-existing call graph construction, points-to analyses and dataflow analyses provided with Soot respective ASM. The bad experiences I had with Soot for Javarifier and other tested purity checkers was one factor to that I didn't look at Soot in any detail until late in my development when I already spent months working with ExtendJ.

The work has been greatly simplified thanks the use of DrAST. DrAST is a visualizations tool and attribute debugger for JastAdd developed as master thesis project Joel Lindholm and Johan Thorsberg 2016 [62]. DrAST provides a visualization of the AST after parsing a given input to a Compiler built with JastAdd. DrAST provides the possibility to inspect the values for the attributes of each of the nodes in the AST and also demand evaluation of the annotation. DrAST does in this way provide debugging by seeing the

values of the attributes at different nodes in the AST.

DrAST helped me quickly gain an understanding for ExtendJs AST structure. The work would have been much slower without this visualisation.

The changes, I had to make to JastAdd internally took me long time to find where the equation was. These I had to find manually giving me a clear view for exactly how hard it is to understand the structure of a large tool without any aid.

11 Related Work

The designed purity checker has a lot in common with JPure attempting to provide fast modular purity checking for use in common development. The annotation provides for methods to be checked almost independently from all the rest. Like with JPure the main information that must cross the method boarders is whether the caller is fresh and the arguments are fresh and if the returned value is fresh.

Since this tool is an extension in addition to the information collected by JPure a few new points of information needs to cross the barrier between methods. The information if a value is cached or not and type of attribute. Alias information doesn't need to cross the border due to "the laws of locality" which Pearce presented. This vastly simplify the analysis but sacrifices precision when more complex changes of objects occur.

I deemed this loss of precision acceptable since for the objects of greatest interest ASTNode are intended to be "almost immutable" and the changes are very local. For general Java, the purity checker approach will of course not accept all programs which are valid. You are not generally allowed by the checker to change object through other objects either.

I decided to separate the concepts of freshness and purity more the JPure did.

The checker maybe should have fully adopted the ideas of Efftp with modification annotation for modified localities and the parametrised returned localities to clear more Java programs. I didn't build in those in the checker since I felt the annotations I had where sufficient for typical JastAdd situations. I partly introduced return localities in form of my *@FreshIf* its only that Efftps are more flexible with fields and not only parameters and caller. Time and complexity constraints where are also important. I didn't have time for implement and test if a more powerful annotation system were sufficiently more useful compared to increased analysis costs.

The contribution of the data group abstraction from OpenJML with outlaw of directly accessing the fields used for caching makes sure that the value can be read pre-calculated and makes it easy to spot methods that uses caching.

12 Future Work

Neither of the 2 tools/extensions are complete or perfect and several changes and extension to both can be made in the in the future.

The annotating JastAdd tool generating the annotations could be optimized so the checker doesn't need to perform as much excess testing as its currently preforming due to an optimization of JastAdd. An attribute might not generate a compute method and this forces the checker then to look over all the JastAdd machinery code. If there always is a compute method containing what needs to be checked then all the code in the JastAdd machinery can be ignored reducing the amount of work the tool needs to do.

Late during my work, i update was released to JastAdd improving JastAdds own annotation to the point that they can provide more information to the checker regarding correct annotations of constructor. The new changes could be incorporated into the

tool. These changes require quite a bit of work to be integrated. Further I considered extracting a list of all the annotation separately instead of having the annotation in the templates to make it easier to integrate with changes to JastAdd.

A major weakness in the checker is that it is external library annotation approach is quite hard to work with and have significant weaknesses. It doesn't completely infer the annotations on parameters nor determine if the method has the same return freshness as the caller. There is no complete prioritisation over the options always return fresh or conditional fresh.

The inference capabilities of JPure is greater since it an annotation tool primary. JPure annotates all methods and parameters fully to fixed point. This purity checker could be extended to preform that as opposed to the single pass solution currently.

It doesn't fully utilize shadowing super types and unbounded captures to reduce the size of the lists of external methods that needs to be annotated. The format is also not pretty especially for methods with many parameters. An unique simpler representation which would have to be separately parsed could make the annotation easier to verify and manually correct. Improved inference capability would make the tool more easily used the current scheme is rudimentary requiring excessive manual verification. Full inference as in JPure could be implemented with some changes.

Java 8 concepts like lambdas and function references are not fully explored I didn't really considered them much and ExtendJ didn't seem to provide the type analysis I need to allow quick implementation of analysis for them. This would probably change with further development of ExtendJ.

Implementation of a more detailed analysis for aliasing and data structures could be included to be used when more precision is desired giving more options to configure the precision used by the checker depending on the time the programmer is willing to spend on the analysis.

Providing better debugging of why additional annotations are suggested could help the programmer understand and find problems during annotation verification which I presented as the fourth step in the use case for the tool in chapter 6.2.

There are opportunities to make the code faster and more readable by sorting away different aspects and improve the caching for the most important variables.

13 Conclusion

The conditions JastAdd imposes are many and where beyond the scope of all the previous existing tools. I designed a purity tool for JastAdd which works for some JastAdd and Java programs. The tool can detect several types of side effects but are in no way perfect. Java is complicated and I have only implemented for a subset of Java. The limitation is mostly in Java 8 features such as lambda constructions for which I have implemented only limited checking other misses can also be present.

The design of the checker was for ease of implementation not for optimal tracing of the origin of a given problem. The modular checking checks within a method what problems might be present and can answer the problem is here with these assignments.

The developer however is interested in which attribute calls the modifying code or why the code is included in the checking. This information is only full available during the annotation iterations. Despite storing data to resolve this the warnings aren't as easy to trace to the root causes as one could hope.

It performs a fast analysis pass over the code which even for large programs will only take time comparable to the compilation time. This is after the more expensive task of generating the additional needed annotations in iterations for the user code and library code which take more time but is done once. The tool can detect a lot of basic programming errors.

I explored at one point in how hard a purity checker is to develop using RAG. Inter declarative programming allows the specification of different functionalities to be compact and in one place and is the main benefit. The interesting analysis steps which is the pointer analysis and the call graph construction. Is effectively implemented using circular attributes and collections. The pointer analysis uses circular attributes on the CFG. Circular attribute allows simple formulation of data flow like analysis.

Warnings are easily collected with collection attributes. The collections attribute provides traversing of such that the checking is performed for every Java construction using very little code to specify it.

In short RAG work, quite well for designing a purity checker the only difference is the memory usage. For example, a freshness analysis which simply moves forward and preforms all lot of field side effects would not have to cache all the intermediary values of the freshness analysis.

The annotations of external library functions are the source of most of the problems for the checker. It takes time to verify and correct the annotation given to external library functions which the checker cannot inspect. The library annotations are a major source of unsoundness since the checker will give them the attribute required to silence warnings. This is the drawback of an analyser which doesn't operate on byte code. The annotations are stored in an inconvenient way in the external files.

References

[1] Arran D Stewart, Rachel Cardell-Oliver, and Rowan Davies. Csse technical report uwa-csse-14-001. Side effect and purity checking in Java : a review. 2014.

[2] Lars R Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[3] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*, pages 9–18. Australian Computer Society, Inc., 2005.

[4] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 82–91. IEEE, 2004.

[5] Torbjörn Ekman and Görel Hedin. The jastadd system—modular extensible compiler construction. *Science of Computer Programming*, 69(1):14–26, 2007.

[6] Jastadds homepage. http://jastadd.org/web/index.php, 2017. Accessed:2017-5-10.

[7] Donald E Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[8] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[9] Eva Magnusson and Görel Hedin. Circular reference attributed grammars-their evaluation and applications. *Electronic Notes in Theoretical Computer Science*, 82(3):532–554, 2003.

[10] David J Pearce. Jpure: a modular purity system for Java. In *International Conference on Compiler Construction*, pages 104–123. Springer, 2011.

[11] Wei Huang and Ana Milanova. Reiminfer: method purity inference for Java. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 38. ACM, 2012.

[12] Matthew S. Tschantz and Michael D. Ernst. Java ri: Adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 211–230, New York, NY, USA, 2005. ACM.

[13] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, et al. Object and reference immutability using Java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 75–84. ACM, 2007.

[14] Néstor Catano and Marieke Huisman. Chase: A static checker for jml's assignable clause. In *Verification, Model Checking, and Abstract Interpretation*, pages 26–40. Springer, 2003.

[15] David R Cok. Openjml: software verification for Java 7 using jml, openjdk, and eclipse. *arXiv preprint arXiv:1404.6608*, 2014.

[16] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Revealing purity and side effects on functions for reusing Java libraries. In *International Conference on Software Reuse*, pages 314–329. Springer, 2015.

[17] Jesper Öqvist and Görel Hedin. Extending the jastadd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 147–152. ACM, 2013.

[18] Jesper Öqvist. Extendj - the jastadd extensible Java compiler, May 2017. http://www.extendj.org.

[19] Görel Hedin. Edan65 - compilers. http://cs.lth.se/edan65, 2016. Accessed:2017-5-10.

[20] Niklas Fors. The design and implementation of bloqqi–a feature-based diagram programming language. 2016.

[21] Sergiy Kolesnikov, Ing Sven Apel, and Christian Lengauer. An extensible compiler for feature-oriented programming in Java. 2011.

[22] The JastAdd group. Jastadds homepage, May 2017.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[24] Torbjörn Ekman and Görel Hedin. The jastadd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, October 2007.

[25] Erik Hogeman. Extending jastaddj to Java8. Master's Thesis LU-CS-EX: 2014-14, Dept of Computer Science, Lund University, 2014.

[26] Friedrich Steimann, Jesper Öqvist, and Görel Hedin. Multitudes of objects: First implementation and case study for Java . *Journal of Object Technology*, 13(5):1–1, 2014.

[27] Emma Söderberg, TorbjöRn Ekman, Görel Hedin, and Eva Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 78(10):1809–1827, 2013.

[28] Manuel Geffken, Hannes Saffrich, and Peter Thiemann. Precise interprocedural side-effect analysis. In *International Colloquium on Theoretical Aspects of Computing*, pages 188–205. Springer, 2014.

[29] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *ACM Sigplan Notices*, 29(6):230–241, 1994.

[30] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[31] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[32] Paul Anderson, David Binkley, Genevieve Rosay, and Tim Teitelbaum. Flow insensitive points-to sets. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 79–89. IEEE, 2001.

[33] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[34] Frank Tip and Jens Palsberg. *Scalable propagation-based call graph construction algorithms*, volume 35. ACM, 2000.

[35] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50. ACM, 2009.

[36] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D Ernst. Ownership and immutability in generic Java. In *ACM Sigplan Notices*, volume 45, pages 598–617. ACM, 2010.

[37] K Rustan M Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. *ACM SIGPLAN Notices*, 37(5):246–257, 2002.

[38] Gary T Leavens and Yoonsik Cheon. Design by contract with jml, 2006.

[39] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in Java . In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174. ACM, 2008.

[40] Haiying Xu, Christopher JF Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007.

[41] David A Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205–224, 2007.

[42] Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java -like Programs (FTfJP)*, 2004.

[43] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[44] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.

[45] Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java -like Programs*, page 4. ACM, 2013.

[46] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005.

[47] Shay Artzi, Adam Kieżun, Jaime Quinonez, and Michael D Ernst. Parameter reference immutability: formal definition, inference tool, and comparison. *Automated Software Engineering*, 16(1):145–192, 2009.

[48] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D Ernst. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 233–269. Springer, 2013.

[49] Jamie Quinonez. *Java rifier: Inference of reference immutability in Java.* PhD thesis, Massachusetts Institute of Technology, 2008.

[50] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D Ernst, Joe Kiniry, Gary T Leavens, K Rustan M Leino, and Erik Poll. An overview of jml tools and applications. *Electronic Notes in Theoretical Computer Science*, 66(2), 2003.

[51] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with sat. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 109–120. ACM, 2006.

[52] K Rustan M Leino. Data groups: Specifying the modification of extended state. In *ACM SIGPLAN Notices*, volume 33, pages 144–153. ACM, 1998.

[53] Greg Nelson. Extended static checking for Java . In *International Conference on Mathematics of Program Construction*, pages 1–1. Springer, 2004.

[54] D Cok and Gary T Leavens. Extensions of the theory of observational purity and a practical design for jml. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008). Number CS-TR-08-07 in Technical Report. School of EECS, UCF*, volume 4000, pages 32816–2362, 2008.

[55] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Towards purity-guided refactoring in Java. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 521–525. IEEE, 2015.

[56] Naoto Ogura, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Hey! are you injecting side effect?: A tool for detecting purity changes in Java methods. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–3. IEEE, 2016.

[57] Type annotations and pluggable type systems (the Java ™ tutorials > learning the Java language > annotations), May 2017.

[58] Jesper Öqvist. Simplecfg extension for extendj, May 2017.

[59] Görel Hedin. An introductory tutorial on jastadd attribute grammars. *Generative & Transformational Techniques in Software Engineering III*, pages 166–200, 2011.

[60] Properties, May 2017.

[61] The JastAdd group. Jastaddtest, May 2017.

[62] Joel Lindholm and Johan Thorsberg. Drast-an attribute debugger for jastadd. *LU-CS-EX 2016-10*, 2016.

Sidoeffektsanalys för referens-attribut-grammatik

Mikael Johnsson
Lunds Universitet
2017

Sidoeffekter

Det finns situationer då det är viktigt att veta att metoder inte har några sidoeffekter. Det går då att genomföra olika optimeringar, bland annat för kompilatorer, utan att orsaka fel.

Avsaknad av sidoeffekter kan visa att metoder inte påverkar resultatet av varandra. Anropsordningen kan då optimeras och delresultat kan memoriseras, det vill säga sparas och återanvändas, vilket annars kan leda till fel. En sidoeffektschecker kan användas för att undvika sådana fel.

Jag har programmerat en sidoeffektschecker för Java specifikt för att användas med JastAdd. JastAdd är ett verktyg för att konstruera kompilatorer med hjälp av en formalism kallad referens-attribut-grammatik. JastAdd ställer krav på olika Javametoder som att de måste vara sidoeffektsfria och producera nya objekt. Dessa krav finns eftersom JastAdd använder tekniker som kräver sidoeffektsfrihet. Ingen verifiering görs av dessa krav vilket lämnar risken för att man gör fel och det är det som jag försökt att åtgärda.

Jag testade först om diverse tidigare verktyg för Java kan användas även för JastAdd. Jag bedömde att ingen av de testade alternativen fungerade riktigt för JastAdd. En flesta klarar inte av memorisering eller de andra specialfallen som dyker upp vid användande med JastAdd. De mest kapabla verktygen är för långsamma och svåra att arbeta med.

Jag började därför programmera ett nytt verktyg som är anpassat efter de krav som JastAdd ställer. Jag designed ett nytt typsystem med annoteringer utformade för att hantera de krav JastAdd ställer.

Mitt verktyg genomför en snabb analys som kan hitta många vanliga fel som nybörjare till JastAdds kan göra. Även för professionella projekt kan sidoeffekter som har introducerats möjligen upptäckas.

Mitt bidrag är ett verktyg som kan användas för att upptäcka vanliga sidoeffeckts fel in Java och som inte är för svårt att manuellt annotera för. Annoteringarna är designade med tanke på svårighet att använda annoteringarna, analyskostnad och vilka olika beteende man troligen vill tillåta. Jag kombinerade tidigare idéer från andra verktyg som JPure och OpenJML tillsammans med nya idéer för att hantera JastAdd-relaterade situationer.

Typsystemschecker

Jag provade flera olika typsystemsverktyg men de hade alla olika nackdelar. JPure som blev min primära inspiration använde egna annotering för sidoeffektfrihet så som *@Fresh* för nyskapande objekt, *@Local* för lokala ändringar och så vidare. JPure's annoteringar kändes nära till beteendena som jag behöver kategorisera metoder i.

Example 1. Olika annoterade metoder.

```
@Secret(group=X) X value=null; int counter=0;

@Pure(group=X) public X method(){
        if (value!=null)
                return value;
        value=calculate();
        return value;
}

@Fresh public X calculate(){
        counter++; //Fel
        return new X();
}

@Local public int count(){
        return counter++;
}
```

Exemplen visar några metoder med annoteringar mitt verktyg använder. Annoteringen *@Local* innebär ändring i samma objekt eller platser nåbara via *@Local* annoterade referenser. *@Pure* indikerar en metod som ska vara sidoeffektsfri förutom hemliga sidoeffekter till fält med *Secret*.

Memoriseringar använder sig av tillåtna sidoeffekter, nämligen att spara delresultaten i fält. Genom att göra dessa fält *@Secret* klarar analysen av att skilja på tillåtna och otillåtna sidoeffekter. Datagrupper kan vidare bestämma var memoriseringen få ske.