Semester Project

Transformation of Java Card into Diet Java Card

Erich Laube
laubee@student.ethz.ch

March 2005

Software Component Technology Group
ETH Zurich
Switzerland

Prof. Peter Müller
Supervisor: Ádám Darvas

At the ETH in Zurich, research is being done on semi-automatic verification of Java code. Part of this research is a tool called `Jive` (Java Interactive Verification Environment). `Jive` only handles a subset of the Java language, called Diet Java Card (DJC). This work describes the transformations I implemented to translate normal Java code into this subset.

Contents

1. Introduction and Motivation

1.1. Document Overview

This document is organised as follows: in the first (this) chapter, I give a brief introduction into the project and the bigger project it is a subpart of. The second chapter is a general overview of the transformations, what possibilities there are and I will also give some pointers as on where to find the transformations in the code. In chapters three, four and five I explain in detail the various transformations. Chapter six is an outlook into future work. It will also list the transformations that still need to be implemented and how the existing transformations could be optimised in order to allow programmers to better benefit from them.

1.2. The Jive Project

The Software Component Technology group at ETH Zurich and the Softwaretechnik Group at TU Kaiserslautern work on an interactive program prover called `Jive` (Java Interactive Verification Environment). The goal is a program which allows one to formally prove that a Java program behaves as specified. The specification language of `Jive` is JML which introduces concepts like pre- and postconditions of methods and class invariants.

The tool aims at the verification of Java Card programs. Java Card is a subset of Java and excludes, for instance, threads and floating point values. To reduce the amount of rules needed for the formal verification, some limitations to the Java Card language were introduced and the resulting, simplified language was called Diet Java Card. It imposes restrictions to the programmer, which mainly are of syntactical nature.

1.3. Language Limitations

The whole verification process of `Jive` is based on first order logic. For every language construct a hoare rule is defined. A module of `Jive` takes the JML specification of the code and creates hoare triples. With the hoare rules, it is then tried to prove the hoare triples and thus the input code. While most constructs could be expressed with a rule, the aim is to have a set of rules which is as small as possible. For instance there is only a rule for `while` loops and no rule for `do` or `for` loops, as they can be expressed in a

`while` loop.

Having a small set of rules allows easier reasoning that these rules are correct, thus improve the certainty that the proofs done by the theorem prover are sound. But programmers who write their code in Java likely will not care much about this and still want to use the constructs they know. This is where the transformations in this paper come in. A precompiler transforms the normal Java code into Diet Java Card code (the part covered by the hoare rules). By this intermediate step, the programmers and the provers can both work on their preferred language set.

1.4. Used Tools

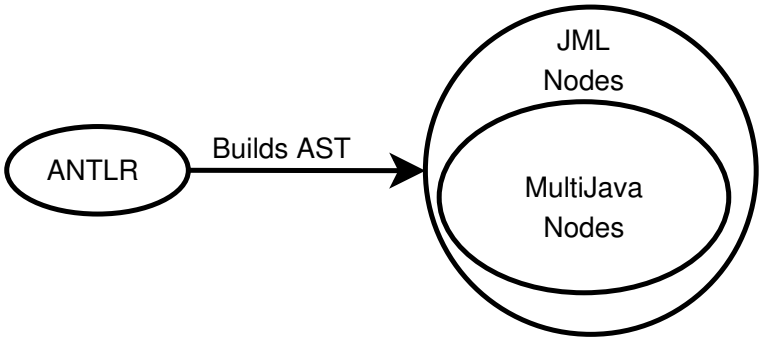

Figure 1.1.: ANTLR, MultiJava and JML

1.4.1. ANTLR

`Antlr` [1], ANother Tool for Language Recognition is a freely available language tool that provides a framework for constructing recognisers, compilers and translators from grammatical descriptions. In this work it is used to parse the structure of the Java source files and produces the AST (Abstract Syntax Tree).

1.4.2. MultiJava

MultiJava [2] is a free Java compiler developed at the Iowa State University. The compiler uses `Antlr` to build the AST and then does all the static checks on it, as well as producing the byte code for the virtual machine. As the source code is freely available, it is possible to write ones own checks for the AST as well as doing transformations on the tree.

1.4.3. JML

Lastly, JML (Java Modeling Language) [3] is a specification language used to specify the behaviour of Java modules. It is built on top of the MultiJava compiler and introduces new node types to wrap around the normal Java node types. From this metadata the specification is read which is needed to create the hoare triples for the verification in `Jive`.

1.5. Goals of this Project

The aim of this project was to allow as many constructs as possible in the input language by transforming them into constructs which exist in DJC. The focus of the project at first was mainly on transformations on the statement level. Later in the project I also started work on expressions, more on those in chapter 5 on page 43.

2. Transformation Basics

2.1. Overview of a Compilation

To better understand the transformations, here a brief introduction into how the three tools used in this project work together. Assume there is a Java file `C.java` to be compiled. Upon invoking the compiler, it takes the plain text file containing the source code and runs a lexer over it. The lexer looks at the file and checks if its layout meets the requirements of an extended backus naur form (EBNF), which defines the Java language.

Once the lexer has done its work and did not find any errors, the parser is called, which goes through the file and creates an abstract syntax tree (AST). The AST consists of nodes, each of which represent a Java construct. These node classes are provided by the MultiJava framework. The important classes for this work are the `JStatement` and the `JExpression` classes with all their subclasses (each have around 30 direct subclasses). These classes are important for the project because they also represent the constructs that need to be transformed.

After the parser has finished, the MultiJava/JML typechecker has its turn. It performs static checks on the AST, calculates types of certain constructs and adds more information to the AST. As a last step, usually the code generator would be called which writes the `.class` file containing the bytecode of the Java class.

The JML part of the compiler is only of limited interest to this project. The JML part adds specification of the code directly into the AST. JML specifications however are written into comments and do not directly influence the code, especially not the parts that are transformed in this project - so, while it is there, and also important for the Jive project, it does not have much to do with the transformations of the source code.

2.2. Finding the Transformations in the Code

There are two levels where transformations take place in this project. One is the `Jive.g` file which provides the parser with information as on how to build the AST. Transformations that do not need type information can be put there. Also, all constructs which cannot be converted and have to stay forbidden can be caught here and cause the compilation to abort with an error. The difficulty in forbidding constructs here is that they also will be forbidden for the JML part, where some constructs, like lazy or '||' are

allowed.

The other place where transformations were introduced is in the compiler, after type checking. The advantage is that the AST is complete and all type information inserted. The disadvantage is that the traversing of the tree has to be implemented explicitly. As the node layout of the AST is not uniform, each node class has to be identified and handled specially, such that a lot of code has to be written.

I have chosen to traverse the AST using recursive methods, which can be found in the file `DJCTransformer.java`. Using a visitor design pattern would probably have been the nicer way of doing it, however there are many nodes which I am not interested in and for using a visitor I would have had to implement handlers for them as well. The other point speaking against a visitor was that most transformations result in a node being split up into several nodes. Attaching them to the AST seemed easier for me to do with recursive methods than with a visitor.

The code of the actual transformations is in the file `JiveTransformer.java`. Some of the transformations on the grammar level also call methods of this class - editing a Java source file is just more convenient than an `Antlr` grammar file.

2.3. Unique Identifiers

As new methods, fields and variables are introduced to the transformed classes, a way had to be found to give them unique names. As there are transformations at the grammar level, where it is not known yet what names future variables of the same block could introduce, building a list of identifiers and using any identifiers but these was not a feasible solution. Instead the idea of adding a prefix was implemented.

There are two approaches to this idea, one would be to add the prefix to all identifiers of the original code, the other to add it in front of the newly introduced ones. As the programmers should still be able to recognise their own code after the transformations, the latter approach was taken. To guarantee uniqueness, a trick has to be used. The prefix that is added has to contain a character which is not allowed in normal Java code. For this project I used the degree sign '°', but any other character which is not allowed for Java identifiers could be used. Identifiers are only checked for validity by the lexer, once in the AST, the compiler does not care anymore how identifiers look like - they are just strings.

In addition to the illegal character, a counter was introduced that increases in value whenever a new identifier is created. In the end, the implementation looks like this:

```
static long runner = 0;
```

```
public static String uniqueIdent(){
  String s = "°v" + runner;
  runner++;
  return s;
}
```

Note that in the code examples I often left the prefix out in order to improve the readability of this document.

2.4. Conventions

In the sections ahead I will use the following conventions. First, I will describe the problem, then go into theoretical reasoning how the construct can be converted. Next I will describe the actual implementation by giving a pseudo code for the algorithm I used. If the generic algorithm has some obvious drawbacks and creates hard to understand code for some instances of the problem, I will explain this in the last paragraphs. For simple, straightforward transformations I will compress this all into one paragraph.

3. Transformations in Jive.g

The `Jive.g` file is the grammar file for `Antlr`, which contains all the transformations I did on the grammar level. `Antlr` supports inheritance among grammar files (Figure 3.1). When the grammar file is compiled, a Java file is created which represents the parser and is included into the compiler. In this project, there are three grammar files which are woven into the final compiler. These are the grammar file for MultiJava, which is inherited and extended by the grammar file for JML from which the Jive.g file inherits and adds the Jive specific parts. When the grammar is compiled, first a file `expandedJive.g` is generated which contains the complete grammar of Jive, JML and MultiJava. In a second step, this is converted into a Java file.

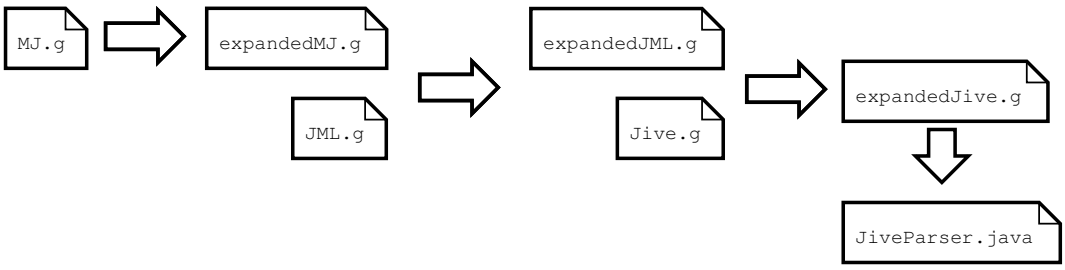

Figure 3.1.: Grammar Inheritance

3.1. Compilation Units without any Type Declarations

Compilation units without type declarations are forbidden in DJC. Only compilation units that have type declarations - interfaces or class declarations are allowed in DJC. While it would be possible to add a dummy type declaration to an empty compilation unit, I do not think anything would be won with this effort. Hence, a check in the grammar file was added and empty compilation units now cause a compile time error.

3.2. Short If

As the people who write the theorem prover aim at a very small set of rules, there is only one rule defined for `if then else` statements - the one including an else. If short `if` statements (without `else`) were allowed, a separate rule would need to be created

to handle those. The transformation for this construct is straightforward - for every `if` statement it is checked whether there is an `else` statement. If there is no `else` statement, an empty statement is added. This was done by modifying the `if` statement rule in the grammar file.

```
if (cond){
  // something
}

if (cond){
  // something
} else ;
```

Luckily, the Java compiler does not complain about unreachable code if it encounters something like this:

```
if (true){
        // something
      } else {
        // something else,
              // but unreachable code
      }
```

If this would be checked in the compiler and cause an error, the transformation above would not work in all cases and special handlers for the conditions `true` and `false` would have to be implemented.

3.3. For Loop

To keep the rules needed for loops small, there only is one loop construct allowed in DJC, the `while` loop. Thus, every `for` loop needs to be translated into a `while` loop.

```
for(initialiser;condition;increment){
  // body
}
```

A `for` loop consists of four parts: an initialiser, a condition, an increment and the actual loop body. All these parts can be empty, except for the body, which needs to be at least an empty statement (';' or '{}'). This means that `'for(;;) ;'` is a perfectly legal Java statement. - On a side note, yes, that `for` loop does not do anything but create an endless loop and likely no one will ever have such a loop in his code. There will be other examples of code in this document which unlikely will ever be written. But the goal of these transformations is to implement transformations that are semantically equivalent to the code given as input. All that can be assumed about the input code is that it is valid Java code - whether it makes sense or not is not the concern of the transformations.

In the initialiser part, local variables are created and can also be set to some value. This initialisation is executed once for the whole loop. The condition is checked at the beginning of every loop cycle and can be an arbitrary expression. Lastly, there is the increment part, which usually just increases a loop counter, but can also hold almost arbitrary expressions (like object creations for instance). The increment is executed at the end of each loop cycle.

Before I go into the transformation, let us compare this to the `while` loop:

```
while(condition){
  // body
}
```

A `while` loop does not have initialisers - initialising variables used in the condition part has to happen in the code before the loop statement. A `while` loop does not have an explicit increment part either, increments need to written inside the loop body. What at first seems to be the same is the condition which is part of a `for` loop as well as a `while` loop. However, for a `while` loop the condition is not allowed to be empty - `'while() ;'` is not a valid `while` loop.

These differences show that conversion of a `for` loop to a `while` loop is not simply a matter of rearranging the nodes of the AST a little. The initialiser part has to be moved outside the loop construct. This means that new local variable declarations are added to the code before the `while` loop. This is an issue, since in the `for` loop, the variables created in the initialiser only exist in the scope of the `for` loop. To get this aspect correct, the whole transformed construct is wrapped into a new statement block:

```
{
  initialiser;
  for(;condition;increment){
    // body
  }
}
```

The increment part is executed at the end of each loop cycle. It can be added to the end of the body statement list. Again, this is not simply a matter of rearranging the nodes of the AST. The increment part can hold several expressions separated by commas, but a list of expressions separated by commas is not a valid Java statement:

```
for(;;i++,j++,k++){}
      // just moving will not work:
      i++,j++,k++; // not valid Java
```

This means that the list has to be converted into a sequence of statements before it is added to the end of the loop body.

Lastly the condition part. As long as it is not empty, it can simply be reused in the `while` loop as condition. If the `for` condition is empty though, a new condition '`true`' needs to be created for the `while` loop.

Algorithm:

1. If the condition is empty, create a new condition: 'true'.
2. Convert increment expression list to a statement list and add it to the end of the loop body.
3. Create a new while loop W with condition and body from step 1 and 2.
4. Create a new statement block B.
5. Convert the initialiser expression list to a statement list and add that to B.
6. Add W to B and put B at the place where the for loop was.

Possible optimisations:

If the initialiser is empty, the transformation would not need to be inserted into a new statement block.

```
{ // a new block
  initialiser;
  while(condition){
    // body
    increment;
  }
}
```

3.4. Do Loop

Because there only are `while` loops in DJC, `do` loops need to be transformed as well.

```
do{
  // body
} while(condition)
```

A `do` loop is very similar to a `while` loop. It does not have initialisers or increments but just a condition. Like in a `while` loop, the condition cannot be empty. The only difference is that in a `do` loop, the body is executed at least once, meaning that before the check of the condition, there already happened one loop cycle.

A naive approach to transform `do` loops would be to write the loop body in front of the actual loop and then go on with a `while` loop:

```
{ // enclosing block
  // body
  while(condition){
    // body
  }
}
```

Note that, as in the body of the loop new local variables can be defined, there needs to be an enclosing block around the transformation to limit the scope of these variables (like in the `for` loop transformation in the preceding chapter). Actually, the way it is done above is not correct, as variables declared in the body before the loop are valid throughout the loop's execution and would result in errors when the typechecker sees the same variable declaration again inside the loop body.

The other problem with this approach is that inside the body there might be a `break` statement. A `break` statement aborts the execution of a loop and puts the control flow to the statement after the loop. However, if a `break` statement appears outside of a loop context, it has no meaning and the code would no longer be a valid Java program.

Hence, the first execution of the loop's body has to be done within a loop as well. There are two obvious options to do this, one is to create a dummy loop around the transformed block above, which is executed exactly once. The other is to introduce a new temporary boolean variable, which guarantees the first execution.

Loops are hard for theorem provers to handle. They either need the input of the programmer (hence cannot be proven automatically) or invariants specified. Because of this, the idea of a dummy loop was dropped.

```
{
  boolean °v1;
  °v1 = true;
  while(°v1 || condition){
    °v1 = false;
    // body
  }
}
```

This transformation led to another problem. In DJC the lazy or '||' operator is not allowed. One could argue that it is possible to take an eager or '|' instead. If that operator is taken though, the condition is evaluated in the first loop cycle as well, resulting in it being executed one time more in the transformed code than in the original one. This can work well in some cases, but also go wrong in others. Consider the following code:

```
class C{
  int f = 0;
  int foo(){ return f++; }
  do {
    // something
  } while (foo()<6);
}
```

If the condition is something of the form described in the example above, the transformation will no longer be semantically equivalent as method `foo` is called one time too many, resulting in the loop being executed one time less than originally intended.

A nice solution to this problem is to reuse the boolean introduced for the first loop execution. This boolean is still set to true before the loop. Instead of setting it to false inside the loop body, it is set to the value returned by the evaluation of the condition. I decided to still leave an enclosing block in to keep the transformed loop together. As identifiers are unique though, that block could be dropped.

```
{
  boolean °v1;
  °v1 = true;
  while(°v1){
    // body
    °v1 = cond;
  }
}
```

Algorithm:

1. Create statements for the creation of a boolean variable B and setting its value to true.
2. Take the loop body. At the end of the statement block, add the assignment 'B = condition'.
3. Create a while loop with condition B and the body from step 2.
4. Create a statement block. Add to this block the boolean creation and assignment statements from step 1 and as last statement the while loop from step 3.

Optimisations:

In the situation where the loop condition is true the programmer might wonder why the transformer adds a dummy variable that is set to true after each loop cycle. This one case could be optimised.

3.5. Multiple Catch Clauses

Multiple `catch` clauses for one `try` block are not allowed in DJC. There can be only one `catch` clause per `try` block. To transform this, new `try` statements are inserted, such that the number of `try` statements matches the number of `catch` clauses.

Illustration:

```
try {
  // body
}
catch (java.lang.IndexOutOfBoundsException e){
  // handler
}
catch (java.lang.NullPointerException e){
  // handler
}
```

Transformed into:

```
try {
  try {
    // body
  } catch (java.lang.IndexOutOfBoundsException e) {
    // handler
  }
} catch (java.lang.NullPointerException e) {
  // handler
}
```

This transformation is straightforward, the only thing one has to make sure is that the exceptions are still caught in the right order.

3.6. Primitive Types

The primitive types `char`, `long`, `float` and `double` do not exist in DJC. This was included into the grammar file, such that an exception is thrown when any of these types are used and the compilation aborts.

3.7. Unsupported Modifiers

The modifiers `native`, `synchronised`, `transient`, `volatile` and `strictfp` are not supported and cause a compile time error.

3.8. Literals

String literals, character literals and floating point literals are forbidden and also cause the compilation to abort with an error.

3.9. Reflection

Reflection is not allowed as it is not allowed in Java Card. I assume this is handled by there not being a class library for reflection, hence the compiler will complain about it anyway. If this is not the case, a check could be added to the grammar file which would throw an exception if it sees the string '`java.lang.reflect`'.

4. Transformations in MultiJava (Statement Level)

While the transformations in the last section were all performed at parsing time, the following steps happen inside the AST after typechecking is done. The advantage of this is that type information is available, which is crucial for many transformations. The disadvantage is that the parsing of the tree has to be implemented explicitly.

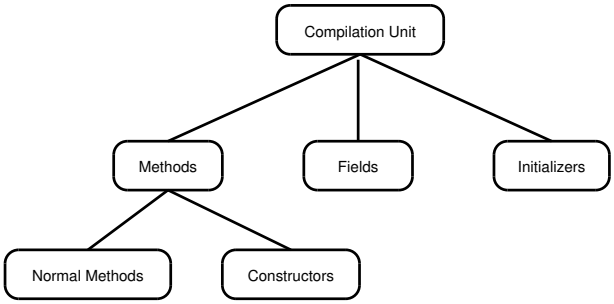

Figure 4.1.: AST Layout

To better understand the transformations at statement level, it is important to know a bit about the AST (also see Figure 4.1). The top node of each AST the transformer ever will see is the compilation unit. A compilation unit node holds references to methods, fields and initialisers. Methods and initialisers then have references to blocks of statements. Inside these blocks of statements, most of the work of the transformer is performed. There are two basic types of nodes to distinguish at this level:

1. Nodes that hold references to more statements - like block or loop nodes, which hold more statements in their bodies.
2. Statements that do not hold further references to statements. These are irrelevant for the further traversion of the tree but may still need to be transformed.

It is important to note that some of the transformations listed in the following sections rely on each other and have to be executed in the right order. This affects mostly the transformation of constructors (which has to be called after transformation of initialisers), but also the transformation of variable declarations (which has to be called as the last transformation after all other transformations have finished).

4.1. Static Initialisers and Fields

```
class Initialiser{
  static int f = 2; // static field
  static int init;
  static{     // static initialiser
    init = 7;
  }
}
```

A static field of a class can be accessed without there being an instance of this class around. This means that everywhere in the code of a Java program, a call to '`Initialiser.f`' might occur and the result of this call should yield the value two. Transforming this is difficult, as I have no control over when the initialiser will be executed exactly, or where the result is stored, as there is no instance of the class.

An idea I had was to introduce a new static `boolean` field b. The default value of any boolean is `false`, hence b would automatically be set to `false`. The initialisers could then be moved into a static initialiser method. Inside this initialiser method, I check the value of b. If b is false, this is the first execution of the initialiser method and initialising needs to be done. The last statement of the initialiser would set b to `true`. If b is true, the initialiser was already executed and nothing would need to be done.

```
class Initialiser{
  static int f;
  static int init;
  static boolean b;
  static void static_init(){
    if(b) {}
    else {
      f = 2;
      init = 7;
     }
  }
}
```

As a next step, each call to a static field would need to be caught and modified in a way that before that call the static init method is called. The static init method would also have to be called when an instance of a class is created. However, when does the initialising of b to its default value (false) happen? This is left to the virtual machine and out of the scope of the Jive verification process. Hence, in the end it was decided to still forbid static fields and initialisers and the compiler will abort compilation at the sight of these.

4.2. Nonstatic Initialisers and Fields

```
class Initialiser{
  int f = 2; // nonstatic field
  int init;
    // nonstatic initialiser
    init = 7;
  }
}
```

Contrary to their static equivalents, nonstatic fields and initialisers can - and are converted. They are tied to an instance of a class and it is clear how the initialisation works. At object creation, first space in memory is allocated and all fields are set to their default values (0, `false`, `null`). Then the constructor of the super class is called, initialisers are executed and finally the constructor body itself is executed.

This means that it is possible to modify the default field values of the super object in an initialiser:

```
class B{
  int f = 2;
}
class C extends B{
  {
  super.f = 4;
  }
}
```

When an instance of class C is created, the execution order of initialisers and constructors is as follows: consObject, initB, consB, initC, consC - hence the value of field f equals four after object creation.

To transform initialisers, a new method is created, I call it `inst_init` (with a prefix for uniqueness). This method is called by the constructor after the call to a `super()` or `this()` constructor. So, each constructor of a class needs to be modified and the call to the inst_init method is inserted after the call to the super method. If a class has no constructor, a default constructor needs to be created.

In the first version of this transformation the `inst_init` method was always created, no matter if there was anything to initialise or not. Later, this was optimised in such a way that it was only created if there actually was anything to initialise.

A tricky thing is that constructors of the same class may call each other via `this()`. As initialisers might have side effects, it needs to be guaranteed that the `inst_init` method is only executed once. To ensure this, a new boolean field 'init' is added to the

class, like in the sketch for a conversion of static instance initialisers. By default, the value of that boolean field is false, which can be used to check if the `inst_init` method was already executed once. In the `inst_init` method the field is set to true. So the `inst_init` method first checks on the boolean, if it is true, it does nothing, if it is false, the initialisation is executed.

```
class B{
  int f;
  boolean init;
  // inst_init method
  void °v1inst_init(){
    if (init){}
    else {
      f = 2;
      init = true;
    }
  }
  B(){ // constructor
    inst_init();
  }
}
class C extends B{
  boolean init;
  void °v2inst_init{
    if (init){}
    else {
      {
        super.f = 4;
      }
    }
  }
  C(){
    inst_init();
  }
}
```

Having added a new method to the class means that this method will also be looked at by the Jive prover. This means that a JML specification has to be provided for the `inst_init` method. This is not implemented yet, as there was no time left towards the end of the project. For the `inst_init` method, the specification would consist of all the fields that were initialised to a value plus the sum of all JML specifications of initialisers, which should already have a specification written by the programmer.

Algorithm:

1. Collect all initialisers and fields with init values. Fields with initialisers are of the form 'vartype varname = initialvalue' - the declaration part needs to be left in the class block, but the initialiser part needs to be moved into the initialiser method. See conversion of variable and array initialisers for details.

2. If the number of collected items equals zero, return.

3. Else, add a new boolean 'init' to the fields.

4. Create a new method, inst_init and put all initialisers in there. Make sure to put them there in the right order. As a last statement add the assignment of init to true.

5. Loop over all constructors and add the call to the inst_init method. If there is no explicit this or super constructor call, the new method call will be the first statement, if there is an explicit constructor call, the new method call will be the second statement of the constructor body.

6. If the class has no constructors, add a new default constructor which only has the call to the inst_init method.

Optimisations:

The introduction of the `init` boolean which ensures that initialising only happens once is superfluous in many cases. For instance if only value types are assigned to fields in the initialisation, it does not matter if this initialisation happens more than once. Checks could be added here, which would make sure that the boolean field is only added if it is really needed. This would improve the readability of the code created by the transformer.

Instead of giving the `inst_init` methods unique names by using the $^{\circ}v_i$ prefix, the method could also be given a `private` modifier, which would allow that always the same name is used. This would create less confusing identifiers for the user. On the other hand side, with the current version the user can recognise every piece of code added by the transformations by the unique prefixes.

4.3. Constructors

```
class C{
  int f;
  C(){
    f = 2;
  }
  C(int i) {
```

```
    f = i;
  }
}
```

In DJC, constructors are not allowed. To create objects, a call like C c = new C() is allowed, but it only allocates space in the memory and sets all fields to their default values. Explicit constructors that do more than that are forbidden. To solve this limitation, for each constructor a new method of return type void is created. To make a unique method name out of the constructor name, the prefix '°c' is attached.

The result of the conversion of the simple example above looks like this:

```
class C{
  int f;
  °cC(){
    f = 2;
  }
  °cC(int i) {
    f = i;
  }
}
```

A tricky thing is that inside a constructor, there is (either explicit or implicit) another constructor invocation - most often the constructor of the super class, but sometimes also a different constructor of the same class via `this()`. This constructor call needs to be found and a call to the new constructor method has to be added. An exception to this are constructor calls to the class `Object`. For this class it was assumed that it will not meet the transformer, hence not have a °`cObject` constructor method.

In Java it is guaranteed that if there is an explicit call to another constructor within a constructor, this call is the first statement in its body. So basically the first statement of each constructor body has to be looked at - if it is a constructor call it is transformed into the corresponding call to the constructor method, if not, a new statement needs to be added in front of the first statement. In MultiJava this is a bit different though. At typechecking time, MultiJava moves explicit constructor calls to a field of the constructor body and replaces the constructor call by an empty statement:

```
// before typechecking
class C{
  C(){
    super();
  }
}
// after typechecking
class C{
```

```
    C(){
        ;
    }
}
```

The advantage of this is that implicit constructor invocations are also created at compile time and moved into that field of the constructor body. The disadvantage is that transformations cannot be done inside that field and as constructors are transformed into methods - which do not have the special constructor body - these calls need to be moved back into the constructor body.

I decided to do this in the instance initialiser transformation out of two reasons. First, as there already is a new method added to the constructor body in that transformation, it is convenient to add the call to another constructor in the same step. Second, if there is no constructor, a constructor is created in the initialiser transformation. This constructor will not meet the typechecker before the constructor transformation, hence the field where the constructor calls would be stored in will not be set (and it cannot be set manually as it is not visible from the code of the initialiser transformation). This would then mean that in the constructor transformation, it needs to be checked if the constructor being converted comes from the initialiser method or was already there in the original source code.

With the implementation of the optimisation that an `inst_init` method was only created when it was really needed, two new problems arised. The first problem was that before the optimisation, the initialiser transformation made sure that there was at least one constructor in the class before the constructor transformation was executed. With the early termination (see step 2 of the algorithm) this no longer was the case. The other problem was that due to the early termination, explicit constructor calls were not moved from the field back to inside the code (last paragraph).

Currently this is solved as follows: the initialiser transformation makes sure that constructor call fields are added back into the body, no matter if there was an `inst_init` method created or not. Whether there are constructors around or not is checked within the constructor transformation. This solution works, but it is not very nice, as now the constructor transformation will not work without a preceding instance initialiser conversion. A better way of doing it would be the creation of a transformation that just puts the constructor calls back into the code, which is executed before the initialiser transformation. This would make sure that the constructor transformation will still work without modifications if in a later version of DJC initialisers were allowed.

There was an argument whether creation of a new constructor method was really needed if there were no constructors defined. After all, setting default values is already done by the 'new' call. For instance consider class D: a subclass of Object which has no default constructor. The default constructor method created will be just empty:

```
// no constructor method needed:
class D{}
// still converted to:
class D{
  void °cD(){}
}
```

This looks pretty useless. However, other classes that use class D, either need to assume that a constructor method exists for D or perform checks if that method exists. This quickly becomes rather complicated if we look at a more complex example:

```
class A{
  int i;
  A(){ i = 2;}
}
class B extends A{}
class C extends B{
  int j;
  C(){ j = 3;}
}
```

In this example, B looks very much like class D in the example above. But B does need a constructor method or the fields of A would never be initialised. It is clear that A needs a constructor method. C needs a constructor method to initialise its own field. While it would be possible to introduce checks for special cases where constructor methods are not needed or special cases where the method needs not be called, it would be rather tedious and error prone to do. By giving every class the constructor method, every class can be handled in the same generic way - at the cost of having some artifacts which would not need to be there.

```
class A{
  int i;
  °cA(){
    i = 2;
  }
}
class B extends A{
  °cB(){
    super.°cA();
  }
}
class C extends B{
  int j;
  °cC(){
    super.°cB();
```

```
        j = 3;
    }
}
```

The other tricky part is to scan the whole code for constructor calls. Each constructor call has to be replaced by the call to the default constructor, followed by the constructor method. This is explained in a later chapter when expressions are described.

As a last point, like the `inst_init` method, the new constructor methods need a JML specification in order to allow verification by `Jive`. Currently, this JML specification is not implemented. The specification would basically consist of the specification for the old constructor, plus the one of the initialiser method.

Algorithm:

1. Remove each constructor C of a class and replace it by a constructor method °cC, with the same body and parameters, but return type void.

2. If no constructors are found, add a constructor method which only calls the constructor method of the super class.

3. For each constructor method, check whether the explicit constructor call goes to a super class named 'Object'. If this is the case, replace the call by an empty statement. If not, replace the constructor call by the call to the corresponding constructor method.

Optimisations:

As mentioned in the example above, constructor methods are not always needed. Calls to the super constructor method are not always needed either. So these could be removed. This would require many tests to be added to the constructor and constructor call transformations though.

Super constructor calls to class Object are currently replaced by an empty statement. This might look confusing to users and could be removed completely. The cost of this optimisation is that the constructor body would have to be created anew as there is no operator to just remove the first statement of a statement block.

There were also some arguments about whether the 'super' prefix in the calls to the super constructor methods were really needed. After all that method name is unique and the method binding should yield the same result no matter if there is a 'super' prefix or not. To me it seems easier for the programmer to understand the code when there is a 'super' prefix. It then is clear by just looking at it that a method of a super class is called. If the user just sees a '°cS' method call, he might wonder what that call is there

for and where it goes to.

4.4. Local Variables

In DJC, local variables can only be declared at the beginning of a block. That block can be any block inside the code and does not have to be the main class block.

In Java, variable names are not unique, local variables can shadow fields of the current class, as well as fields of the super class. This means that variable declarations cannot just be moved to the top of their block.

To perform the transformation, every block needs to be investigated. If a variable declaration is found, it needs to be checked if it shadows any class variables or if it is a new one. If the latter is the case, it can be moved to the top of the block, if not, a new block is inserted. The rest of the code will always need to be put into the innermost block, as the variable declaration is valid till the end of the old block.

Finding out if a local variable shadows any fields or not has its difficulties. To find all visible fields of a class I implemented a method which recursively goes upwards in the type hierarchy and adds field names to one container and type names to a second container. The second container makes sure that each type is only checked once for its fields. Into the first container all found non-private fields are inserted. Lastly, the private fields of the current class need to be added as well, because they are visible for the class methods and can be shadowed. The private fields of super classes are of no interest for this transformation, as these cannot be accessed anyway.

```
class localvars{
  int i = 2;
  void block(){
    System.out.println(i);
    int k;
    System.out.println(i);
    int i = 4; // shadows class field i
    System.out.println(i);
    // more code
  }
}

class localvars{
  int i = 2;
  void block() {
    int k;
    System.out.println(i);
    System.out.println(i);
```

```
    { // new block
      int i = 4;
      System.out.println(i);
      // more code // needs to be put into this block
    }
  }
}
```

Algorithm, move variables on class level:

1. Once for the class, find all fields visible for the methods of the class.

2. For every method body of the class call the moveVariables transformation.

Algorithm `moveVariables`, takes a statement block and a list of all visible fields as parameters:

1. Iterate through the statements of the block. Keep a container for variable declarations and one for other statements.

2. For each statement:

   - If it holds more statements, i.e. is a block of statements, call `moveVariables` on that block.
   - If it is a variable declaration and does not shadow any fields, put it in the variable declaration container.
   - If it is a variable declaration and does shadow a field, start a new block and put the declaration at the beginning of that block. From now on, the rest of the code also needs to be put into this block, as the variable needs to be valid till the end of the original block.
   - Else, put it in the other container.

3. At the end, create a new block. Put the content of the variable container at the top of that block (in the correct order). Then insert the content of the other container, also in the right order, into that block.

As many transformations introduce new temporary variables, this transformation is called two times, first before any transformations on the AST take place and then after all other transformations are done. This has the effect that in the end, the topmost variable declarations of a block are those of the programmer and the newly introduced variables appear afterwards.

Optimisations:

Consider the following code:

```
class C{
  int f;
  void foo(){
    // code that does not touch class field f
    int f; // local variable, shadows f
    // more code
  }
}
```

The transformation described above would create a new block upon reaching the declaration of the local variable f. However, as the code in method foo does not access the class field f, this new block is not required. Checks could be implemented to eliminate unnecessary blocks. This would improve the readability of the code. The check would need to iterate over all possible paths of executions though and would be tedious to write.

4.5. Fields and Variable Declarations

It is not allowed to declare several local variables or fields at once in DJC:

```
class C{
  int a,b,c; // without initialisers
  void foo(){
    int d=2,e=1,f; //with initialisers
  }
}
```

They have to be declared one by one. Note that there might be initialisers to these variables, which complicates the transformation a bit. Apart from that, the transformation is straightforward.

4.6. Array Initialisers

```
int[] a = {1,2,3,4,5};
```

In DJC, array initialisers are forbidden. This means they need to be picked apart and rewritten into separate statements:

```
int[] a;
```

```
a = new int[5];
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
a[4] = 5;
```

This basically does not cause any difficulties, all information is available inside the node with the array initialiser. Due to the preceding transformations array declarations with initialisers are guaranteed to only occur inside of methods. This is important as the transformation cannot take place inside a field body, because statements are not allowed there.

Note that this transformation actually happens on two separate levels. Once when initialisers and fields are converted into the inst_init method and the other time when the transformation is recursively applied to all methods of the class. It would be possible to handle the inst_init method in the later run, too, but then temporarily something like

```
class A{
  int[] a;
  void inst_init(){
    a = {1,2,3,4,5}
  }
}
```

would be in the code, which is not valid Java code. It would also mean that the transformation would have to explicitly search for assignments where the right part is a list of expressions.

4.7. Variable Initialisers

```
int a = 3;
int b = foo();
```

Variable initialisers are not allowed in DJC. Like with array initialisers, declaration and assignment need to be picked apart and written into separate statements:

```
int a;
a = 3;
int b;
b = foo();
```

This conversion is straightforward and also was executed for fields when they were converted.

4.8. Switch Statements

`Switch` statements are forbidden in DJC. A `switch` consists of the following elements:

```
switch(expression){
  case 1:  // code 1
  case i:  // code i
  default: // code default
  case j:  // code j
  case n:  // code n
}
```

An expression is evaluated at the top of the block and its result is then tested for equality at each `case` label. If a label is found that matches the result of the expression, the code behind that label is executed. Once a match has been found, there are no more checks for equality and the code behind label i - assuming i is the matching label - is executed. After that code is executed, the code of the next label is executed, unless there is a `break` or return statement somewhere in the code of label i. The code of the default label is executed if either there was no match at all or if the code of the label before the default label was executed and did not end in a `break` or `return` statement.

It is important to note that if no matching label has been found upon reaching the default label, still first all other labels are checked for a match. The code inside switch labels can be almost arbitrary, there can be further `switch` statements nested in switches, there can be loops, variable declarations, etc.

Another important thing to note is that variable declarations inside a case label's code affect the whole remaining part of the `switch` block:

```
switch(i){
  case 1:
    int j = 1;
    break;
  default:
    j = 2;
    System.out.println(j);
}
```

If the `switch` expression i is not equal to 1, the default label's code is executed. The assignment of j to 2 is valid, which means that the declaration of that variable in the

code of `case` 1 was executed even though basically the code of `case` 1 was not executed as there was no match. What was not executed is the assignment of j to 1 in `case` label 1. This causes some difficulties for the transformation.

As `break` statements are allowed within `switch` block, the converted `switch` either has to be enclosed in a `while` loop or all `break` statements have to be removed and replaced by `if then else` constructs. As loops are impossible to prove without specified invariants, I was urged to pursue a solution without loops. However, `break` statements can be nested deep inside the code of a `case` label. So the solution I went for is to separate difficult `switch` statements and simple ones.

The requirements for a `switch` statement to be considered simple are the following:

- There either is no break statement or it is the last statement of a labelled block.
- There either is no default label or the default label is the last of all labels.
- There are no variable declarations inside the switch block. Variable declarations inside statements that hold other statements are fine.
- There are no switch statements inside the labelled blocks.

The first point is the most important, as tracking the `break` statements and transforming them into a different construct with the same semantics is the issue here. I am sure that given enough time and effort, a generic transformation without `while` loop for all `switch` statements could be implemented. But as the time for this project was limited and there were other important transformations to do, I left it at this. The current requirements still allow loops, `try-catch` and `if` statements inside the `switch` block.

Another thing to be aware of is that the `switch` expression might have side effects. It is hence important to ensure that it is evaluated only once. This is achieved by introducing a temporary variable that stores the result of the `switch` expression. That variable then becomes the new switch expression.

An example of a 'simple `switch`':

```
switch(foo()){ //  simple switch
  case 1:
    System.out.print(1);
    break;
  case 2:
    System.out.print(2);
  case 3:
    System.out.print(3);
  default:
```

```
    System.out.print(-1);
}

{ // transformed switch:
  int °v1;
  °v1 = foo();
  if (°v1 == 1) {
    System.out.print(1);
  } else if (°v1 == 2) {
    System.out.print(2);
    System.out.print(3);
    System.out.print(-1);
  } else if (°v1 == 3) {
    System.out.print(3);
    System.out.print(-1);
  } else if (true) {
    System.out.print(-1);
  } else ;
}
```

Algorithm 'simple switch':

1. Create a temporary variable v of the switch expression's result type and assign the result of that expression to it.

2. For each label except the default label: create an expression E_i that checks for equality with v. For the default label use the boolean literal true as expression.

3. For each code block behind a label, if the last statement is not a break statement add the code of the following blocks until a break statement is reached or all labelled code blocks are traversed. Once a block with a break statement is reached, add the block, but remove the break statement. Let us call the created blocks B_i.

4. Build a nested if then else statement with the following layout: if (E_0) B_0 else if (E_1) B_1 .... else if (true) B_n else ;.

5. The easiest way to implement the creation of the tree is to start at the bottom, with the default label and work the way upwards in a loop. The if then else statement that was created in the last cycle always becomes the else block of the next cycle until finished.

6. Enclose the whole transformation into a block to guarantee that the temporary variable is only visible in the scope of this transformation.

Optimisations:

It was brought to my attention that I could just do 'else B_n ' for the default statement instead of 'else if (true) B_n else ;'. This is of course true as long as the switch that is

being converted is not of the following form:

```
switch(expr){
  default:
    // some code
}
```

Again, it is unlikely that anyone will ever write this code, but it is allowed Java code, hence needs to be supported. To catch this special case, extra checks would need to be inserted, for which at the moment was not enough time - the focus lied on doing many transformations and not doing few but optimise them the best way.

The next example shows a difficult switch. It has a variable declaration inside a case label and a `break` statement enclosed in an if block. As `break` statements can be used, code blocks are not copied for transformation of difficult switches.

```
switch(foo()){ // complicated switch
  case 1:
    int m;
    break;
  case 2:
    m = 2;
    System.out.print(m);
  default:
    System.out.print(-1);
  case 3:
    if( bar() > 3) break;
    System.out.print(3);
}

{
  boolean °v1;
  °v1 = true;
  while (°v1) {
    int °v2;
    int m;
    °v1 = false;
    °v2 = foo();
    °v1 = °v1 | °v2 == 1;
    if (°v1) {
      break;
    } else ;
    °v1 = °v1 | °v2 == 2;
    if (°v1) {
      m = 2;
      System.out.print(m);
    } else ;
    °v1 = °v1 | °v2 == 2;
```

```
    if (°v1) {
      System.out.print(-1);
    } else ;
    °v1 = °v1 | °v2 == 3;
    if (°v1) {
      if (bar()) break;
      else ;
      System.out.print(3);
    } else ;
    if (°v1) ;
    else {
      System.out.print(-1);
      if (bar()) break;
      else ;
      System.out.print(3);
    }
    °v1 = false;
  }
}
```

Algorithm 'complicated switch':

1. Create a boolean b, set it to true and make a while loop with b as condition. (This is to handle break statements.)

2. In the while loop's body: set b to false, create a variable v, set it to the evaluated switch expression.

3. For each label, create an expression E_i that checks equality of v and the label case. For the default label the expression E_i is the expression E_{i-1}. This emulates the fall through behaviour for labels without break statements.

4. For each label: execute $b = b|E_i$, if b is true, execute the label's code. Do not forget to add an empty statement for the else clause. Once the default label is traversed, keep adding up code labels into a separate block B. This block is needed for the execution of the default label.

5. At last, insert a runtime check that looks if any label has been executed. In the else case add block B that now contains all code that needs to be executed at the default statement.

6. Set b to false to make sure the loop is not executed again.

Optimisations:

In the first version of this transformation, I did not reuse the temporary boolean b and kept a growing list of or statements at the if statements that replaced the case labels. (if $(E_i|E_2|E_3....)$). As this looked irritating, I started reusing the boolean to temporary

hold these expression values. The cost of this is that the transformation logic itself has become a bit more complicated to understand because the boolean variable is used for two completely different things.

Like in the transformation for simple switches, a `switch` statement with only the default label is not optimised:

```
switch(foo()){
  default:
  if( bar() ) break;
    System.out.print(-1);
}

{
  boolean °v1;
  °v1 = true;
  while (°v1) {
    int °v2;
    °v1 = false;
    °v2 = foo();
    °v1 = °v1;
    if (°v1) {
      if (bar()) break;
      else ;
      System.out.print(-1);
    } else ;
    if (°v1) ;
    else {
      if (bar()) break;
      else ;
      System.out.print(-1);
    }
    °v1 = false;
  }
}
```

5. Transformations in MultiJava (Expression Level)

Expressions in general turned out to be rather complex to transform. This is due to the following properties of expressions. First of all, expressions can be arbitrary nested, which makes tracking complicated. There are even more different types of expressions to track down than it was the case with statements. The other problem is that most transformations work by introducing additional statements. Hence, a way needs to be found to put in code at the statement level while working inside expressions.

The approach I have taken to tackle the first problem is again the use of recursive methods which go through all the nodes of the tree. One method goes through statements and recursively calls itself if it finds statements which hold other statements. For any found statement, another method is called which finds out whether the node holds references to expressions. If this is the case, a transformer method is called on the expression, which recursively transforms the expression and all referenced expressions.

Most problematic expressions are of the following form:

```
foo(a, b, exprWithSideEffect);
```

This means that there is a node inside the expression tree which has a side effect and this is not allowed in DJC. The solution is to assign the expression with a side effect to a temporary variable and work with that instead:

```
c = exprWithSideEffect;
foo(a, b, c);
```

Thus it needs to be possible to introduce new statements before the statement that held the expression which causes problems. To deal with this problem the method that simplifies expressions '`simplifyExpression`' holds references to two containers. I called them 'before' and 'after'. Into these containers I put the new statements created by the transformation. It is the callers responsibility to correctly put the statements stored in the containers into the AST and also emptying the containers before reuse. The third parameter is a flag, which roughly indicates where the call to the method comes from. Four different cases are distinguished:

- Right hand side (RHS) of an assignment to a local variable at topmost level (not nested assignment).
- The expression is the expression of an expression statement.
- The expression is an expression of any other statement (such as a return statement).
- The expression is part of another expression.

The `simplifyExpression` method returns a new expression, which is meant to replace the expression it just simplified. Putting this together, the signature of the method is this:

```
expression simplifyExpression(
  expression exprWithSideEffect,
  container before,
  container after,
  int calledFrom);
```

Let us take a look at an example of a transformation. Consider this code:

```
foo(bar());
```

The statement above represents an expression statement node (let us call it S) in the AST. Its expression is a method call (E_1: foo(...)), which as parameter takes another method call expression (E_2: bar()). When S is reached, `simplifyExpression` is called on E_1 with two empty containers and the flag indicating that the call comes from an expression statement. So far everything is fine as the constructs are all allowed in DJC. The expression transformer now takes a look at the expressions referenced by E_1 - which are the parameters of the method call, in this case E_2. `simplifyExpression` is called on E_2, the flag now indicates that E_2 is part of another expression, the two containers are still empty.

The method sees that E_2 is a method call and within an expression. This is not allowed. Now it first tries to simplify all expressions referenced by E_2 - in this case there are none. Then it creates a local variable V (of the same type as E_2) and puts it into the 'before' container. The next statement it creates is an assignment of V to E_2 , which also is put into the container. Then it returns the reference to V to the caller.

The caller is the `simplifyExpression` that is handling E_1. It replaces the parameter 'bar()' by the returned expression and then returns the modified expression E_1.

Now we are back at the statement level, where first all elements of the 'before' container are inserted into the AST, then the current statement is added and lastly, all elements of the 'after' container.

After transformation:

```
T temp;
temp = bar();
foo(temp);
```

This is the basic layout of most of the transformations ahead, which are just different cases the `simplifyExpression` method has to handle. What might seem unclear at the moment is the use of the 'after' container. I will elaborate on that in the section about post increments.

5.1. Correct Handling of the Before and After Containers

When `simplifyExpression` returns, it is the callers responsibility to correctly handle the two containers and put the statements in, or forward them to its caller. Sometimes expressions that would be allowed in DJC need to be put into the 'before' container as well to ensure correctness. This can be the case in expressions that hold references to more than one expression. Consider the following code:

```
class C{
  int f = 0;
  int foo() { return f++; }
  void bar(){
    parameters(f,foo(),foo());
  }
  void parameters(int a, int b, int c){}
}
```

In this example, method `foo` returns an integer value and also modifies the state of the object, by adding 1 to the field f. In the method `bar` there is a call to `parameters`, which takes three parameters. The method call node holds three references to expressions - one expression per parameter. The transformations of parameter two and three add statements to the 'before' container, as these are method calls. However, as these method calls have side effects, the first parameter needs to be saved as well.

If the parameter would not be saved, the statements added by the transformations, which are written into the 'before' container and executed before the call to `parameters`. This would result in the following code (simplified):

```
// f not saved:
void bar(){
  v1 = foo(); // return 0, f = 1
  v2 = foo(); // return 1, f = 2
  parameters(f,v1,v2);
  // or for better readability:
  parameters(2,0,1);
```

```
}

// f saved, correct version:
void bar(){
  v0 = f;
  v1 = foo(); // return 0, f = 1
  v2 = foo(); // return 1, f = 2
  parameters(v0,v1,v2);
  // or for better readability:
  parameters(0,0,1);
}
```

As can be seen, the first transformation is not correct, but the second one is. This means that in an expression which references more than one expression, sometimes value type parameters have to be saved even though they would be allowed in DJC, because the other expressions can change the values of these parameters.

Currently, the way this is implemented is as follows. If a transformation of one of the expressions in the parameters list add content to the 'before' container, all parameters are stored into separate variables. This is a bit overkill though and could be optimised. First of all, the problem described above only applies to value types - reference types cannot be saved anyway, as the saved reference would just point to the same object, which is modified by the method with side effects.

Optimisations:

A first optimisation would thus be to only save value types. This would already remove much unnecessary code. In a second step, the statements put into the 'before' container could be further investigated.

5.2. Expressions with Side Effects

In DJC, the left hand side of an assignment where the right hand side has a side effect can only be a local variable. This is exactly solved in the way described in the last two sections. For an expression with a side effect a new temporary variable is introduced, assigned to that expression and then used in the expression:

```
a = foo(b) + bar(c);

// transformed into:
int °v1;
°v1 = foo(b);
int °v2;
°v2 = bar(c);
```

```
a = °v1 + °v2;
```

As the expression transformer was the last piece of the project I implemented, it currently does not cover all expressions with side effects. What is handled are `field-accesses`, `array-accesses`, `method invocations` and `object creations`. Conversion of DIV and MOD are currently not implemented but should not cause further difficulties.

5.3. Assignments within Expressions

```
a = b = c = d;
```

Assignment within expressions is not allowed in DJC. In assignments like these, `d` is evaluated and the result assigned to `a`, `b` and `c`. The last element of the assignment list can be an arbitrary expression. This means that `d` can have side effects. As it is not guaranteed that `a`, `b` and `c` are local variables (assignment of expressions with side effects is only allowed to local variables), a new temporary variable needs to be created to which `d` is assigned to. The value of this local variable is then assigned to `c` and handed further to `b` and `a`.

The transformation is straightforward. If the expression transformer is called on an assignment expression with the flag not set to 'RHS of assignment to local variable at topmost level', the assignment is put into the 'before' container and the left hand side of the assignment is returned to the caller.

5.4. Compound Assignment

Compound assignment operators (†= where † represents any binary operation) are forbidden.

```
i += 2;
j <<= 5;
```

The transformation of this construct also is straightforward, the operator †is taken away and a new assignment expression is created where on the right hand side the mathematical operation is explicitly performed:

```
i = i + 2;
j = j << 5;
```

To be completely correct with this transformation, a typecast of the RHS expression to the LHS type would need to be inserted. This is automatically taken over by the MultiJava compiler.

At the beginning of the project where it was not sure yet if I was going to work on expressions, this transformation was only executed on the statement level - compound assignments within expressions were not handled. When the expression transformer was implemented the old method was integrated into the new one.

5.5. Casts

In DJC, casts may not appear within expressions. It is only allowed to assign the result of a cast to a local variable. This was also implemented within the expression transformer - a new temporary variable is created, the result of the cast is assigned to it and then that variable is used.

5.6. Keyword new for Arrays

In DJC, the keyword `new` may only be followed by a type name and a pair of brackets containing an expression (which is evaluated to a number) for array creation. As multidimensional arrays are handled on the grammar level, the only thing to make sure in the expression transformer is the handling of anonymous arrays in parameters or return values. This means that new temporary variables have to be introduced whenever anonymous arrays are created. This is again done by introducing a new temporary variable of the needed array type, assigning the result of the array creation to it and then passing it as parameter.

5.7. Keyword new for Object Creation

Creation of new objects is only allowed by using the keyword `new` followed by the class name and a pair of empty parenthesis. It is not allowed to put arguments into the parenthesis as there are no constructors.

By introducing constructor methods, there now is a workaround for this restriction. However, all constructor calls need to be converted as follows:

```
// before transformation:
A a = new A(p1,p2);
// after transformation:
A a;
```

```
a = new A();
a.°cA(p1,p2);
```

This is also implemented using the 'before' container. Anonymous objects given to methods as parameters or to return statements as return values are also solved like this.

Word of warning: this transformation may only be called once for the whole code. In the example above, it cannot be seen if 'a = new A()' is the original call to the constructor as the programmer wrote it, or if it is already transformed. By looking at the surrounding statements, i.e. searching for a call to the constructor method, this could be checked, if there ever arises the need to have the transformation executed twice. But at the moment, no such checks are in the code.

5.8. Array Accesses within Expressions

Array accesses may only occur at the outermost level of a statement, outside of an expression. This is checked using the flag indicating where the call comes from in the `simplifyExpression` method. If the requirements are not fulfilled, the access is moved into the 'before' container where it is assigned to a temporary variable, which will then be used in the expression.

5.9. Pre- and Postfix Operators

In DJC, pre- and postfix operators are not allowed. While these seem easy to convert (after all it is just altering a variable's value by plus/minus one), they are rather tricky. A postincrement adds the value one to a variable after the current expression has been executed but before the next expression is executed. This means that for a transformation on the statement level the execution of an expression has to be interrupted to execute the increment.

Consider the following code:

```
int i = 0;
i = foo(++i) + foo(i++) + foo(++i);
```

Calculating the values for the increments, the code that actually is executed is the following (assuming that foo does not change the value of variable i).

```
i = foo(1) + foo(1) + foo(3);
```

Execution of statements before an expression is done using the 'before' container. For postincrements, there is the 'after' container. Basically, in each transformation it would have to be checked if the transformation led to the 'after' container holding new statements. If this was the case, the current state of the execution of an expression would need to be saved, an increment or decrement statement inserted and the execution would continue with the new variables. This could mangle the code quite severely.

This is just a sketch of how postfix expressions could be handled, as time was running out, I decided not to handle pre- and postfix operators at the moment. To at least allow them in loops, where they are used most extensively, I do allow pre- and postfix expressions as statements on their own. This means if an expression statement which just consists of a pre- or postfix expression is found, it is translated to an assignment $'i = i - 1'$ or $'i = i + 1'$. Used in this way, it does not matter if it is a prefix or a postfix expression and the conversion is also easy to perform.

5.10. While Loop Conditions

A special case for the expression transformer are `while` loop conditions:

```
while(condition){
  // body
}
```

If `while` conditions would be treated like any other expressions, a `while` condition with side effects would be transformed like this:

```
{
  boolean temp;
  temp = condition
  while(temp){
    // body
  }
}
```

Obviously, this is wrong, as the condition only is evaluated once - and not in each loop cycle. Hence, whenever the expression transformer reaches a `while` loop it is transformed into the following form. (I again decided to enclose the whole transformation in a new block to prevent it from being split up by other transformations.)

```
{
  boolean temp;
  temp = condition;
  while(temp){
    // body
```

```
      temp = condition;
    }
  }
```

An exception is loops that formerly were `do` loops, which were transformed into `while` loops in an earlier transformation. These loops already have the right form. This is checked in the transformation. The transformations is again enclosed into a new block, more thoughts on that in 6.2.2 on page 55.

6. Future Work

6.1. Remaining Transformations

6.1.1. Multi Dimensional Arrays

In DJC arrays only have one dimension. Multi dimensional arrays are caught and cause a compile time error. It would be possible to transform multi dimensional arrays into one dimensional ones. A lot of overhead would be generated though, as every array access would have to be rewritten, including calculation of new offsets. This would also have a severe impact on the readability of the code and reduce the chance that programmers can still recognise their code in the transformed one.

6.1.2. Labelled Statements, Labelled Breaks

Labelled statements and labelled breaks are not supported in DJC. I do not think that there are generic conversions possible to address these limitations.

6.1.3. Continue

`Continue` statements are not allowed in DJC. This construct was given very low priority for the project, so not much thought and time was spent into it. It might be possible to transform this statement into a series of `if then else` statements. The benefits of supporting `continue` would in my opinion be rather low, as it might become hard to still recognise the code after the transformation is done. Hence, a check was included and `continue` now causes a compile time error.

6.1.4. Inner Classes

```
class Outer{
  class Inner{}
}
```

Inner classes were given very low priority as well, so I did not get to investigate them much. To me there seemed to be no obvious transformations for inner classes which would keep the semantics. Taking methods out of the inner classes and put them into

the outer class would break encapsulation as well as cause problems with shadowed variables. Putting them into a whole new class at the same level as the outer class would result in semantics being broken as the inner class could no longer access the outer class.

At the end of the project I got to know that the Java virtual machine does not know of inner classes either, hence there already happens a transformation within the Java compiler to make inner classes work. I did not get the time to take a closer look at the tricks used in this progress, but it would surely be an interesting aspect to look into for future projects.

For the moment, inner classes are forbidden and raise a compile time error.

6.1.5. Division and Modulo within Expressions

Division and modulo operators inside of expressions are only allowed at the outermost level. These can be handled in the same way as assignments within expressions. I however, did not have the time to implement these two transformations.

6.1.6. Non-Strict Operators

Non-strict operators &&, || and ? : are not allowed in DJC. All three could be transformed into `if then else` statements. The transformation would likely reduce readability of the code a lot though. Due to time limits, these transformations were not implemented either.

6.1.7. JML Specification for Introduced Methods

In the transformations, currently three new methods are created - these are the inst_init, the block$ and the constructor methods. To be able to properly verify these, a JML specification would need to be provided. This is currently not done. A sketch of what would need to be in those specifications is given in the sections of the respective transformations.

6.1.8. Class Field Expressions

```
class A{
  int a = 2;
  void foo(){
    System.out.println((new A()).a);
  }
}
```

When I tried to convert expressions with field access like the 'new A().a' in the example above I ran into the problem that I did not seem to be able to create a field access expression. There is also an example in the code which shows more exactly what is going wrong. At object creation, certain important fields are not set - looking at the code shows that this can only be done during typechecking. As I did not have the time to figure out how to let the typechecker run over a single expression, I decided to move on and not spend a lot of time here.

6.2. Optimisations

In this section I am presenting some approaches to optimisations which would mainly make the life of the programmer easier by increasing the readability and similarity of the generated code compared to the input code. The list is by no means complete and just holds some generic pointers. I will not repeat the optimisations I mentioned in the sections where I described specific transformations, this part is meant for optimisations that apply to the whole code.

6.2.1. Reuse of Temporary Variables

Most transformations create temporary variables to store results of expressions. Currently, these variables are used exactly once. If one would keep a 'global' index of all these variables, they could be reused and the code would become more compact.

6.2.2. Removal of unneeded Blocks

This optimisation goes hand in hand with the last one. At the moment, most transformations on statement level are enclosed in extra blocks. There were two reasons for this, one was to help indicating what parts of the code belong together and the other reason was to keep the introduced temporary variables valid in only the limited scope of the transformation. The other reason only affects the variables generated by `for` loop initialisers. These variables have to be gone after the loop's execution, as they are only valid inside the scope of that loop. The temporary variables introduced by the transformations have unique names, so it does not matter if they exist throughout the whole remaining lifetime of a class or not. By removing these unneeded blocks, the code would become more compact and temporary variables would also be available for reuse.

6.2.3. Improving the Prettyprinter Class

In MultiJava and JML there exist several classes for printing the AST. Unfortunately, none of them work, which seems to be a known issue. For this project I extended one of

these printer classes and directed the output to the console. It is however not formatted - every line of code starts at the leftermost side of the console. To be able to give nicer feedback to the user, this class should be improved to better format the code.

6.3. Second Typechecking

At the end of the project it was attempted to let the typechecker run a second time over the modified AST. This works now without errors, however it took some hacks to convince JML to typecheck the AST again. These hacks are listed here and should be converted to something nicer in the future to enable the Jive project to do without changes in the MultiJava and JML code.

6.3.1. The block$ Method

It seems that the compiler generates something like an inst_init method already for instance initialisers. It calls this method `block$`. The method is hidden from the user and I only stepped over this when the compiler complained about it not being there at second typechecking. When constructors and initialisers are converted, the whole method container of the class is recreated. This means that the hidden `block$` method is removed in that process. It is however not even wanted to exist anymore after the transformations are done, because in DJC, all we want the empty constructor call to do is allocating space and setting values to (0,`false` and `null`) - and not execute an instance variable initialiser like that `block$` method. This is at least what the code led me to believe the method does.

The hack introduced to keep the compiler from complaining about the absence of this method was to put a `block$` method inside the code. This method just contains the empty body, returns void and does exactly nothing:

```
void block$() {}
```

It might be worth pursuing further what the method exactly is meant to do and if there is a possibility to hide the new method mentioned above again.

6.3.2. Arrays

Array nodes hold a reference to a node of type `JArrayDimsAndInits`. There is a private field `CArrayType` which holds the type of the array (for instance 'int[]'). This field is not set at object creation but later, at typechecking - or at least it should. However, this does not seem to work correctly. The typechecker adds one dimension too much when it resolves types a second time. What I did to fix this was to set the visibility of

`CArrayType` to `public` and manually set the field to the right value. If the value is set, the typechecker is not interested in it anymore and does not change anything.

By introducing this fix the MultiJava source code had to be modified, which is not good as the Jive project aims to use an unmodified version of the MultiJava compiler and not put changes in there with every new version that comes out. What would need to be done is to investigate why things go wrong and find a better fix.

6.3.3. Field Initializers

In MultiJava, a field node is only a wrapper for a variable node. When the transformer removes initializers from variables, the initializer is set to `null`. This seems to cause problems when applied to fields. One solution would be to just recreate the field, but at the time when this conversion takes place, there is no direct reference to the class node to replace the field. The other difficulty with this approach is that the order of field declarations could get mixed up - which is not really a problem anymore once initializers are removed though.

The approach I have taken is to add a method `setVariable` to the field class with which the variable reference the field holds can be reset. (And thus also the initializer value, which is part of the variable). This is again a modification of the JML and MultiJava source code. Hence, closer investigation of what goes wrong would need to be done and a way would need to be found to fix the problem more elegantly.

7. Conclusion

In this document I described the transformations I implemented to allow the programmers more freedom in chosing their prefered language constructs when writing code they want to have verified by the Jive tool. In Appendix A on page 63, the old list of unsupported constructs can be found, in Appendix B on page 67, I put the new, updated list of unsupported constructs. The constructs found on the later list are now all checked and cause compile time errors with meaningful messages.

The transformations as they are now are not very optimised with regard to readability of the code. If the programmer extensively makes use of complicated constructs, such as `switch` or nested expressions, it might be hard for him to recognize his code again once it is transformed. The ideas for future work presented in chapter 6.2 on page 55 could improve this aspect significantly.

Bibliography

[1] Homepage of Antlr, http://www.antlr.org/

[2] Homepage of MultiJava, http://multijava.sourceforge.net/

[3] Homepage of JML, http://www.cs.iastate.edu/~leavens/JML//index.shtml

[4] Java Language Reference, http://java.sun.com/docs/books/jls/

A. Old list of unsupported Constructs

Outside of Type Declarations:

1. Import statements. As all names have to be fully qualified, imports are not necessary.

2. Compilation units without any type declarations.

In Type Declarations:

3. Static and nonstatic initializers.

4. Constructors. This eliminates calls like `this()` and `super()` as well; they are substituted by calls to the appropriate replacement methods.

5. Inner classes.

6. Qualified `this` and `super` (i.e. *Primary*`.this` and *Primary*`.super`) as these are only used in the context of inner classes.

Methods:

7. The old and deprecated method declaration format which, if the method returns an array type, allows one to place the empty brackets of the return type after the formal parameter list, is not supported. (This item is not strict as it only influences the allowed concrete syntax. It can be added if desired.)

Statements:

8. Local variables can only be declared at the beginning of a block. They can be declared within each block, not only at the top of a method body.

9. Short if is not supported, i.e. we always need the full version with `else`. If the second branch is not desired, it can be filled with an empty statement (i.e. just a semicolon) or left empty (the parser should handle this correctly in either case).

10. For-loop, do-loop, switch-block.

11. Multiple catch clauses for one try-block, i.e. each try-block is followed by at most one catch-block.

12. The left-hand side of an assignment where the right-hand-side has a side-effect (i.e. is a field-access, array-access, method invocation, object creation, DIV or MOD operation) can only be a local variable.

    Access of fields via this and super is not critical because this and super are guaranteed to be $\neq$ null.

13. Statement expressions that are just expressions without assignment. The topmost operator of a statement expression must be an assignment, whereas there must not be any other assignments in the RHS expression.

Expressions:

14. Casts may not appear within expressions. There is a cast statement that allows to apply a cast to an expression. The result is assigned to a local variable.

15. The keyword `new` may only be followed by a type name and a pair of brackets containing an expression (which is evaluated to a number and which is allowed to cause a null pointer exception) for array creation or by a classname and a pair of empty parentheses for non-array object creation. It is not allowed to put arguments into the parentheses as we do not have constructors.

16. Array access within expressions. Array access may only take place at the outermost level of a statement, outside of an expression, similar to casts (see above).

Expression Operators:

17. Compound assignment operators ($\dagger=$ where $\dagger$ represents any binary op)

18. Assignment within expressions (`x = y = z`). Expressions must be free of side-effects.

19. Division and modulo operators inside of expressions. For this, we provide special statements which perform division and modulo operations at the outermost level.

20. Pre- and postfix operators ++ and $--$.

21. Non-strict operators &&, $\|$ and ? : .

General Language Constructs:

22. Primitive types char, long, float, double.

23. Multi-dimensional arrays. Arrays may only have one dimension.

24. For variable/field/parameter declarations of array types, it is not allowed to have the brackets at the variable name but only at the type name. (This restriction is not required; the parser can decide this.)

25. Array initializers.

26. Variable initializers.

27. Labeled statements, labeled break.

28. The modifiers native, synchronized, transient, volatile, strictfp (which are not part of Java Card anyways).

29. continue

30. It is not allowed to declare several local variables or fields at once (e.g. `int a, b;`); they have to be declared one by one.

31. String literals, character literals and floating point literals.

32. Reflection (i.e. `.class`) as this is not supported in Java Card.

B. New list of unsupported Constructs

Outside of Type Declarations:

1. Compilation units without any type declarations.

In Type Declarations:

2. Static initializers.
3. Inner classes.
4. Qualified `this` and `super` (i.e. *Primary*`.this` and *Primary*`.super`) as these are only used in the context of inner classes.

Statements:

5. The left-hand side of an assignment where the right-hand-side has a side-effect (i.e. is a field-access, array-access, method invocation, object creation, DIV or MOD operation) can only be a local variable.

   This is partially solved in the transformer. Array access, method invocation and object creation are now fully supported. Field access and DIV or MOD operations are still limited.

   Access of fields via this and super is not critical because this and super are guaranteed to be $\neq$ null.

Expression Operators:

6. Division and modulo operators inside of expressions. For this, we provide special statements which perform division and modulo operations at the outermost level.
7. Pre- and postfix operators $++$ and $--$.

   Pre- and postfix operators are now allowed as statement expressions, i.e. the way they are normally used in loops. Within expressions they are still forbidden.
8. Non-strict operators &&, $\|$ and ? : .

General Language Constructs:

9. Primitive types char, long, float, double.
10. Multi-dimensional arrays. Arrays may only have one dimension.

11. Labeled statements, labeled break.
12. The modifiers native, synchronized, transient, volatile, strictfp (which are not part of Java Card anyways).
13. continue
14. String literals, character literals and floating point literals.
15. Reflection (i.e. `.class`) as this is not supported in Java Card.