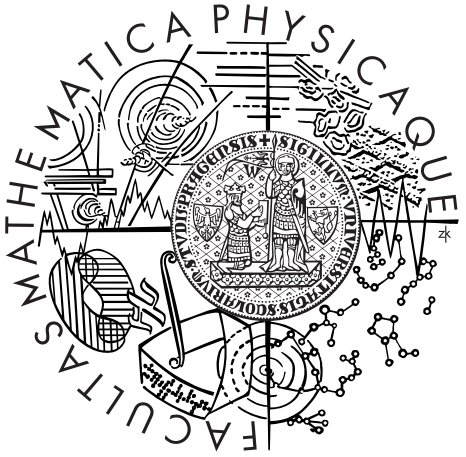
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Petr Malý

Static analysis of C# programs

Department of Software Engineering

Supervisor of the master thesis: RNDr. David Bednárek, Ph.D.

Study programme: Software systems

Specialization: Software engineering

Prague 2014

On this place I would like to express my gratitude to RNDr. David Bednárek, Ph.D. for the supervising of this diploma thesis and his valuable advices. I also want to thank to Mgr. Michal Brabec for his professional opinion. I am also grateful to my family and friends for their patience and support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague 31.7.2014 Bc. Petr Malý

Název práce: Statická analýza programů v C#

Autor: Petr Malý

Katedra: Department of Software Engineering

Vedoucí diplomové práce: RNDr. David Bednárek, Ph.D., Department of Software Engineering

Abstrakt: Cílem této diplomové práce je prozkoumat a aplikovat jednotlivé metody statické analýzy C# programů přeložených do Common Intermediate Language. Výsledky této práce jsou zakomponovány do systému ParallaX Development Envrionment. Tato diplomová práce se zaměřuje na Structural, Points-to a Dependence Analysis.

Klíčová slova: Statická Analýza, C#, ParallaX, *Bobox*, Structural Analysis, Points-to Analysis, Dependence Analysis

Title: Static analysis of C# programs

Author: Petr Malý

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D.,

Abstract: The goal of this diploma thesis is to study and implement selected methods of static code analysis for C# programs translated into the Common Intermediate Language. The results of this work are integrated into the ParallaX Development Environment system. This diploma thesis focuses on Structural, Points-to and Dependence. analysis.

Keywords: Static Analysis, C#, ParallaX, *Bobox*, Structural Analysis, Points-to Analysis, Dependence Analysis

Contents

1. Introduction

This work deals with a static analysis of C# programs. It is a part of the ParallaX development environment which allows a programmer to create a program using the *Bobox* framework. The *Bobox* framework is a complex parallelization environment, which allows to fully utilize multicore processors. This work builds on the results of the master thesis [6] written by Michal Brabec and continues to develop selected parts of the ParallaX development environment. This work concerns algorithms of the static analysis, which are essential for the development of further parts of the ParallaX development environment. The more detailed architecture is described in Chapter 2.

The ParallaX development environment obtains a code written in C# as an input. It is processed in several phases and a C++ code is produced. Then it serves as an input for the compilation with *Bobox* framework.

The code generated into C++ is compiled with the *Bobox* framework to produce a final executable program. The programmer writes a single-threaded code. The *Bobox* framework hides the entire synchronization from the programmer.

The development of a C# program is mostly easier and more comfortable than the development in other languages of the C family. The development is less error-prone. For instance, the use of the *Garbage Collector* and the variable references stops from making memory leaks[1]. The C# type system and language constructs prevent more errors. A great number of errors are detected while the program is compiled into the *CIL*[2]. You can find more reasons in [13].

The proccess of transforming the C# code to a C++ code is not so simple and several issues must be solved, because these languages are not the same and there are several differences. This work provides the results of several analyses to the ParallaX code generator, which produces a C++ code.

Although it is not difficult to generate a C++ code which could be compiled, the final program struggles with performance issues. One part of the ParallaX development environment is an optimizer which uses the results of the analyses studied in this work. The optimizer uses the provided information to create a code which can run parallel.

[1] We assume a program written in a code marked as the *safe code* and not *unsafe*, where the pointers on memory are allowed.

[2] Common intermediate language is bytecode used as an input for Just-in-time compiler.

Only selected static analyses are dealt with in this work. This thesis is concerned with the *structural analysis*, which is part of the *data-flow* analysis, the *points-to* analysis and the *dependence analysis*. The first named analysis is required by two others. It deals with an identification of basic control structures which are then used to unfold control dependences in *dependence analysis*. The *points-to* analysis provides information about dependences in a dynamically allocated memory. This information is supplied to the code generator and the *dependence analysis*. This analysis enables us to detect the parts of the code, which can run parallel. Separate chapters contain a detailed description of the use of all results.

This work is divided into a few chapters. The chapter **Architecture** contains the architecture of the system which the current work is included in. This chapter also consists of the definition of goals determining this work. It contains a detailed description of the topic and the purpose of this diploma thesis.

The chapter **Architecture** is followed by several separate chapters. Those are the chapters **Intermediate Language**, **Control-Flow Analysis**, **Points-to analysis**, **Dependence Analysis**. Each of them begins with a detailed analysis of problems of the topic and then an algorithm description follows. A short conclusion appears at the end of each of those chapters. It also contains a final impact of the analysis on the development of remaining parts of the system.

The description of an intermediate language which is a fundamental part of this work is included in the chapter **Intermediate Language**. It also contain a description of the *structural analysis* which is connected with the intermediate language.

The chapter **Control-Flow Analysis** consists of an *control-flow analysis*. In this chapter the *Structural analysis* algorithms are discussed thoroughly.

It referes to the dynamically allocated memory and references to that memory.

The chapter **Points-to analysis** consists of an analysis of the same name. It referes to the dynamically allocated memory and references to that memory.

The *Dependence analysis* is included in the chapter **Dependence Analysis**. The *control dependences* are thoroughly treated in it.

2. Architecture

In this chapter you can find a detailed description of the context which contains this work. We define main goals and challenges. Related work concerning concerning the studied topic is handeled in this chapter.

2.1 System Description

As mentioned in the previous chapter this work is part of a larger system. The context of the larger system is illustrated in Figure 2.1. This work is part of the ParallaX development environment (or simpler the ParallaX environment).

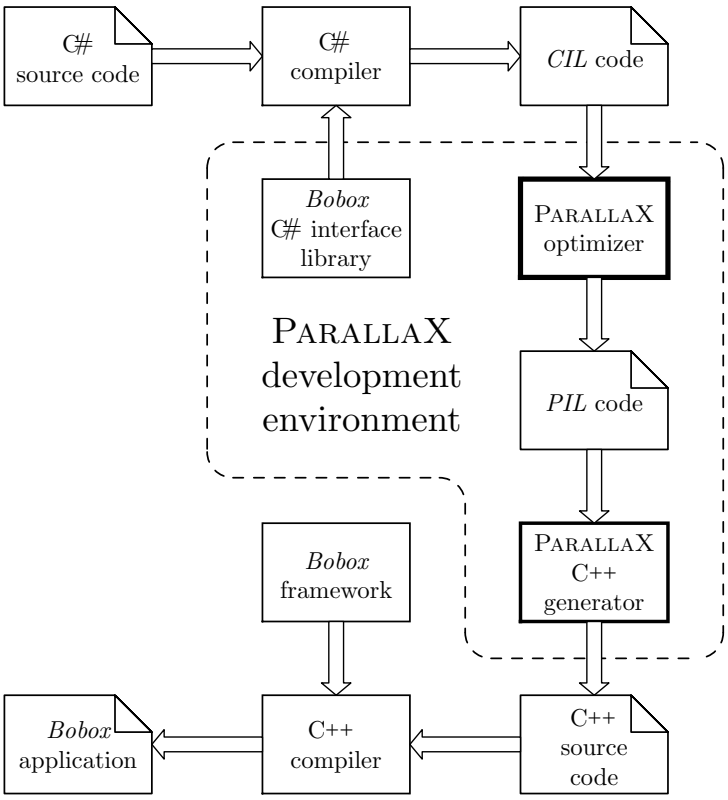


Figure 2.1: Architecture of the ParallaX Development Environment

The ParallaX development environment deals with a preparing the input for *Bobox* framework. The *Bobox* framework is a complex parallelization environment, which enables to a programmer to fully utilize multicore processors. It accepts a C++ code as an input and produces an executable program. You can find more details about the *Bobox* framework in [5].

2.2 ParallaX Development Environment

The main purpose of the ParallaX development environment is to provide a tool, which allows programmer to write a C# program for the *Bobox* framework. It optimize the code and tries to apply an automatic parallelization.

The ParallaX development environment consists of two main parts, which are depicted in Figure 2.2. The first one is an optimizer and the second one is a code generator. The input of ParallaX optimizer is a compiled C# program and its job is to do analyses and transformations required by the code generator. The code generator generates a C++ code, which is in the subsequent stage compiled with the *Bobox* framework to produce an executable program.

The straightforward generation of the C++ code is a routine task and a correct code is created. However this code must be modified so that the available computing resources would be fully utilized and the performance of the final program would be improved.

The development of the ParallaX environment was started by Michal Brabec as a part of the work [6]. He implemented the code inliner which belongs to the ParallaX optimizer. This work concentrates on the phase of code analyses, which is followed by the transforming of a code and the generating of a new one. These operations are not parts of this work and are intented as a further work.

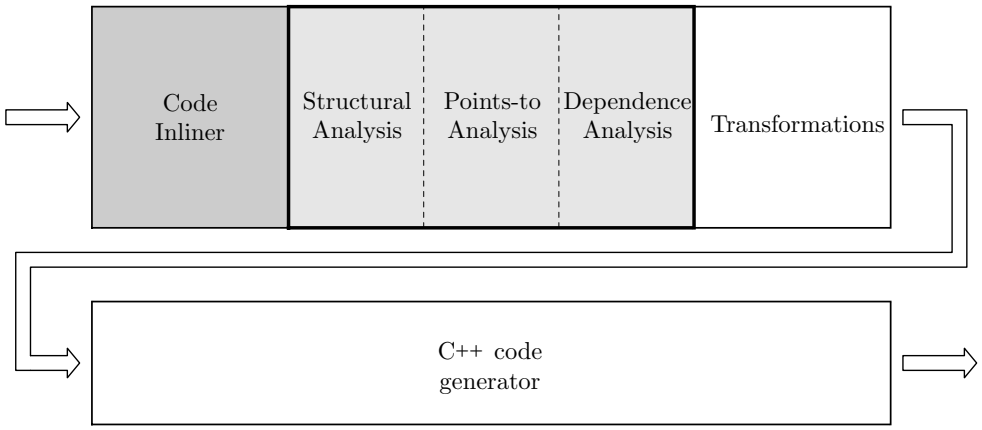


Figure 2.2: Stages of ParallaX optimizer. This work concerns the stages in the bold frame.

2.3 Goals

In order to transform C# programs to a C++ code, we need to track the dynamically allocated memory. It is required because of absence of *Garbage Collector*[1] in C++. Our goal is to do a *points-to analysis*. This analysis provides enough information about the use of the dynamically allocated memory and helps to supply the garbage collector functionality. It enables to study the dependences in that memory, hence an automatic parallelization could be performed in the future.

The second goal is to do a *dependence analysis*. Control and data dependences are obtained from this analysis and together with the results from the *points-to* analysis it makes possible to detect the parts of the code which can run parallel. The automatic parallelizations is made in the stage of transformations, which is not part of this work.

Both analyses will get a method as an input. This method has all method calls recursively inlined with the exception of library calls. Recursion is not permitted. The use of the method which has all method calls inlined is defined in the previous work [6] and has several consequences. Each analysis is applied to a method which can be very large. The points-to analysis can be affected by a low precision of the results under certain circumstances (described in more detail in Chapter 5). The third goal arises from these properties. We group the instructions of the input code together in basic blocks and identify control structures. This task is included in the *structural analysis*. The results of the structural analysis will be also used in the *dependence analysis*, while the control depences are searched for.

Structural analysis, *Points-to set analysis* and *Dependence analysis* are thoroughly described in the further separate chapters.

2.4 Global Problems Analysis

At the beginning writting of this work, the ParallaX optimizer (see Figure 2.2) included only the code inliner. The code inliner accepts a *CIL* code method, the inline expansion is applied to this method. The output of the code inliner is *CIL* code again.

Any work with the *CIL* code is clumsy, because the code is adapted to compiling by *JIT* compiler. It is not suitable for being analysed and transformed by

[1] *Garbage Collector* registers allocated memory and also frees it when it is no more used.

an optimizer. There are too many instructions. A large number of them are only variants of the others and differ only in the operand precision. The transformations include a lot of routine and error-prone work afterwards. The passing of the values through the stack is not convinient for any code transformation. We will design an intermediate language, because of these difficulties. Its purpose will be to simplify the analysing and transforming of the code. The more precise analysis of the designed intermediate language is in Section 3.1 of Chapter 3.

We will not support all the features of the C# language. The features includes the exception handling, unsafe code and anonymous functions. The *Bobox* framework is intended for scientific computations concerning the big data processing. This framework does not contain the exception handling and recovery from it. If an exception is thrown it offen means that the input program is incorrect or the input data are wrong, therefore recovery form the exception is useless. It is possible to implement the rest of the restrictions, but the implementation is hard and needs a large amount of effort. In addition it would not bring any benefit in the expected areas of the use.

In following chapters we will study the existing algorithms of the specified analysis. The algorithms are often applied to the code in the form of an intermediate language. The intermediate language code is obtained by transforming the code which is written in C language or Fortran. The intermediate language could reflect the original language in some aspects, hence it is neccessary to adapt the existing algorithms so that they could be used on the code obtained from the C# code.

2.5 Related Work

The *structural analysis* algorithms are treated in [10]. The algorithms use different algorithms from [1]. These algorithms are well known, hence we can also find them in other publications.

The *points-to analysis* algorithms were first mentioned in the works [3] and [12]. These two publications present two different approaches in obtaining points-to sets. The results of the first approach is more precise than the ones of the second approach. On the other hand the first approach is slower than the second one. There are also publications which combine both basic approaches to obtaining results which are parameterized by a level of accuracy, for instance [11]. The

points-to analysis in [3] is applied on the *C language* code. There are some articles dealing with the point-to analysis in Java ([8], [9]) and fewer articles concerning C# ([4]). In [4] an existing points-to analysis is extended and the inter-procedural aspects of the points-to analysis are dealt with (functions are annotated). We analyse only one C# function, because we have inlined all function calls in the whole function, therefore our work will differ from it. In addition we intend to do a points-to analysis of intermediate language methods.

There are two major groups of dependences - *data-dependences* and *control-dependences*. The first group is very wide and many algorithms exist. Each of them could find only some type of dependences if a certain set of conditions is satisfied. There are many publications for each algorithm. For purposes of this work we will use the algorithms presented in [10] and [2].

3. Intermediate Language

3.1 Problem Analysis

A process of compilation C# programs consists of translating program from C# language to *CIL* bytecode[1]. This bytecode is then used to run the application. *JIT*[2] compiler takes care of translating it to native code of target machine architecture.

Main purpose of *CIL* is to provide code which is platform indenpendent and could be fastly translated into various native codes by *JIT* compiler. Because of this requirement the *CIL* is adapted in a suitable way.

For instance, the instruction for the local variable loading is in *CIL* presented in several forms. The first one in the most general form is `ldloc`. This instruction gets an index as its operand and stores the content of local variable onto the stack. All the other forms has same name, but different suffixes, which indicates its operand or values on the stack. The second form is `ldloc.s` which also requires index of local variable as its operand. The only difference is in the index range. The index of `ldloc` is the two bytes number and the index of `ldloc.s` is one byte number. The other ldloc instructions has no operand and a local variable index is encoded in the instruction's suffix (its binary code). These are `ldloc.0`, `ldloc.1`, `ldloc.2`, `ldloc.3` instructions.

The instructions, which are offen used, have much more shorter code than others. It is suitable for fast analysing by *JIT* compiler and also it spares the size of the output assembly.

The *CIL* design of many instruction variants is not suitable for further analysing or transformations, because each analyse or transformation would have to include all variants of each instruction and this manner would lead to many mistakes.

Local variables in *CIL* are indexed by numbers beginning with 0. This number is then used as a operand in some instructions. The method's arguments are numbered in the same way. There are special instructions[3] for loading of arguments content. They have also suffixes which replace an index in the operand. If we want to modify arguments of method (i.e. remove or add argument) or modify

[1] Common Intermediate Language bytecode
[2] Just In Time compiler
[3] `ldarg`, `ldarg.s`, `ldarg.0`, `ldarg.1`, `ldarg.2` and `ldarg.3`.

the set of local variables, we may face problem of arguments (or local variables) indices renumbering. The indices of variables may change, therefore operands of instructions need to be updated and some instructions also need to be replaced by anothers.

The indices renumbering means an additional work for the programmer. It has same pitfalls as common *CIL* transformations.

In general any small change in *CIL* bytecode often leads to other change, in order to preserve executable code. According to mentioned problems and many others it is required to design intermediate language which supress pitfalls specific for *CIL* and allows to make transformations more comfortable.

We will call the designed intermediate language ParallaX *intermediate language* in the following paragraphs. We will often use a shorter form *PIL*.

3.2 Goals

The main goal of the ParallaX *intermediate language* in context of ParallaX optimizer is to create comfortable tool for a developer, who creates the analyses and the optimalizations upon *CIL* bytecode.

It is not demanded to create the implementation of an executable machine for ParallaX *intermediate language*. The meaning of *PIL* execution will be clarified later in this chapter. It is expected that code will be transformed into another code after all transformations and analyses will be done in ParallaX intermediate language. More details about this transformation are described in Section 3.9.

3.3 Language restrictions

We will use *CIL* code which was generated by C# compiler from Microsoft Visual Studio 2012 as input for compiling to intermediate form. There are several conditions which we will impose on C# code and *CIL* code created from it.

We will not support exception handling, which would complicate development of ParallaX optimizer. ParallaX optimizer is only proof of concept therefore we leave the implementing of this construct as a future possible extension of ParallaX optimizer.

For the same reasons we do not support lambda expressions and anonymous functions.

This diploma thesis uses the results of work [6], where the *CIL* code was analyzed during preliminary phase (more in [6]) and all inner calls of analyzed method were inlined. The inline expansion cannot be done on recursive function, hence the ParallaX intermediate language will be restricted on analysing single method. However calling other functions will be able via special instructions.

3.4 ParallaX Intermediate language requirements

We impose several requirements on ParallaX intermediate language. Following lines briefly express important points of ParallaX intermediate language:

- Simplification of the instruction set
- Ability to add or remove local variables or function arguments
- Leaving the stack design and supplying it with other structures
- Designing a type system
- Code generation to C++ language

All of these points are important and they should not be neglected. If we pay no or only little attention to them, serious troubles may appear in later phases of developing ParallaX optimizer. In consequence we may spend a lot of time on solving problems, which may seem to be easy at the first sight.

3.5 Instruction Set Simplification

We recall an important fact that was mentioned at the beginning of this chapter: there are many variants of each instruction. They are distinguished by suffixes to describe different size of operand or another property.

The reasons of large amount of instructions are the code size and the effective code execution including compilation by *JIT* compiler. These reasons are not important for designed ParallaX intermediate language, because it is not a final form of code and it is expected that other code will be generated from designed intermedite language.

Not all *CIL* instructions might be covered by ParallaX intermediate language. One reason is that we use code generated from the restricted C# code. The second reason is that some instruction are connected with *CIL* type system called *CTS*[4]. Primarily those are the conversion instructions like `conv.i`, `conv.u4`, `conv.ovf.i8` .... These intructions are not used in the ParallaX intermediate language, because of its simplified type system which is described later in Section 3.8.

The original code is not possible to trasform back into the identical form, because the transformation simplifies it. That is why we cannot denote it as a bijection.

For instance in ParallaX optimizer it is possible to transform ParallaX *intermediate language code* back into the *CIL* code. The final code is not identical as the original one, but the meaning of the program remains. Each *PIL* instruction is connected with one from *CIL* (could be more than one). Some *CIL* instructions could not be directly translated into the *PIL* and has to be connected to other *PIL* instructions. An example of these can be `conv`. It is suitable to connect these instructions to others which generate converted value (i.e. `ld` instructions and operation instructions `add`, `mul` etc.).

3.6 Variable Manipulation

We need to get off the *CIL* instruction variants which have encoded the operand inside itself. This could be done by a simple code filtering. The forbidden instructions are searched and replaced by its more general variant. This is not difficult operation and we classify it as a code preparation modification, therefore this modification is in ParallaX optimizer implemented in code preparation phase.

3.7 Stack Problem

While we study *CIL* code we do not know anything about values on the stack without context and simulation of stack work. Let consider following *CIL* code:

```
1 ldc.i4.3
2 stloc.0       // int a = 3;
3 ldc.i4.4
```

[4]Common Type System

```
 4 stloc.1       // int b = 4;
 5 ldloc.0
 6 ldloc.0
 7 mul
 8 stloc.0       // a = a * a
 9 ldloc.1
10 ldloc.1
11 mul
12 stloc.1       // b = b * b
13 ldloc.0
14 ldloc.1
15 add
16 stloc.2       // int result = a + b
```

If we move instructions on lines 9-10 before line 5 we get the following code which has same meaning as previous code:

```
 1 ldc.i4.3
 2 stloc.0       // int a = 3;
 3 ldc.i4.4
 4 stloc.1       // int b = 4;
 5 ldloc.1
 6 ldloc.1
 7 ldloc.0
 8 ldloc.0
 9 mul
10 stloc.0       // a = a * a
11 mul
12 stloc.1       // b = b * b
13 ldloc.0
14 ldloc.1
15 add
16 stloc.2       // int result = a + b
```

From the previous code it is not clear which instructions produce stack values for `mul` instruction on line 9. If we study code thoroughly, we discover that those instructions are on lines 5-6. These instructions do not have to be immediately before `mul` in contrast to first code sample.

We cannot see the track of the stack values in printed *CIL* code without exactly knowing what kind of values does the instruction expect or produce on the stack.

ParallaX optimizer parses *CIL* code with Cecil library and this library does not offer any functionality which would help us to solve this complication. We have to remove the stack and supply the stack values passing with something else.

We have to express passing values somehow. We have the guarantee that all values which were produced on the stack will also be consumed, because the stack must be empty on the end of method execution. If the stack was not be empty the execution of that code would throw an exception of invalid code. This case we do not take into account.

In *CIL*, there is no correlation between how many values from the stack are consumed and how many values are produced. Each instruction (except call instructions) has fixed number of consumed values from the stack and fixed number of produced values. Call instructions get a variable number of arguments from the stack. It also stores return value back on the stack if method returns any value, otherwise nothing is stored onto the stack. Instructions, which do not push any value on the stack, are mostly stores instructions and control instructions (i.e. branches, calls).

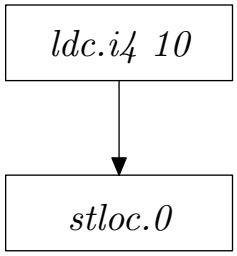
We have to find the way how to express a transfer of value, which has been pushed onto the stack and poped afterwards. We will regard the value transfer as a link between two instructions. We can imagine the whole situation as an oriented graph where the vertices are instructions and the edges are the value transfers.

For example, we can see storing a value of 10 into variable with index 0 in Figure 3.1a. Of course, if there is an edge between two instructions, the instruction, the edge starts at, must be executed before the instruction, the edge ends at. In the figure the instruction `ldc.i4` pushes the constant value of 10 on the stack and then the instruction `stloc.0` pops the value from the stack and stores it into the local variable with index 0.

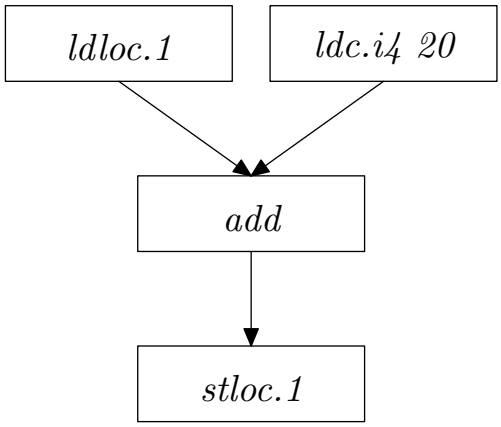
Another example is depicted in Figure 3.1b. The instruction `add` gets two values from the stack, namely the value produced by instruction `ldloc.1` and also value from the instruction `ldc.i4`. We cannot deduce whether the `ldloc.1` will be executed before `ldc.i4` or vice versa. However we can only say that `ldloc.1` will run before `add`, `ldc.i4` before `add` and `add` before `stloc.1`.

We would specify the graph as a rooted tree[5] if there were no instructions which produce more than one value. The instruction `dup` pops a value from the stack, pushes it with its duplicate back onto the stack. A demonstration of this instruction is illustrated in Figure 3.2. The graph in the figure is not a rooted

[5]The rooted tree is a graph that is a tree (we ignore the orientation of the edges) and there is a directed path from each vertex to the root. The root is vertex which has no outgoing edge.



(a) Storing constant value to a local variable



(b) Adding 20 to a local variable

Figure 3.1: Stack values forwarding graph

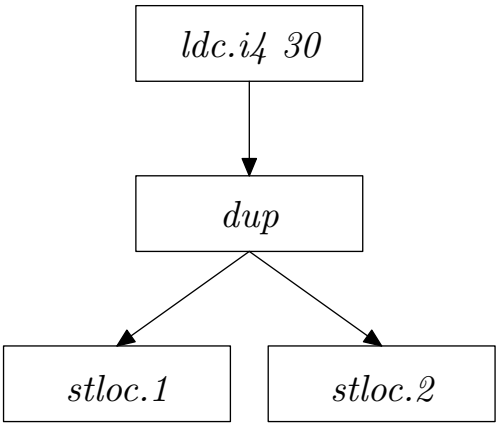


Figure 3.2: Storing the same constant value to local variables with indices 1 and 2

tree because there is no directed path from `stloc.2` to `stloc.1` or vice versa.

In Figure 3.3 the `dup` instruction also breaks the definition of the graph (there could at most one edge between two vertices). We can denote the passing of stack values as DAG[6] if we allow graphs to be multigraphs. The graph[7] cannot contain a cycle because if there were any cycle, it would not be possible to make a correct execution order of the instructions.

Mostly there are only instructions which push only one value onto the stack, but we cannot rely on that and we must take the instructions similar to `dup` into a considiration.

[6]Directed Acyclic Graph

[7] In the following text we will understand a graph to be a multigraph.

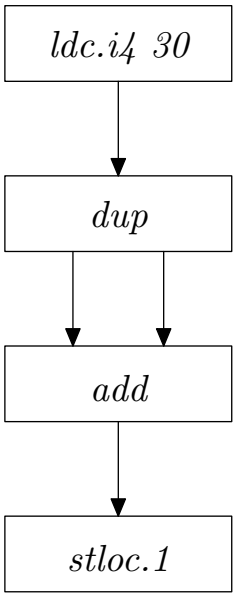


Figure 3.3: Storing the sum of two constant values 30 to a local variable

3.8 Type System

The code written in *CIL* byte code uses *CTS*[8], which is part of *CLI*. *CTS* is designed to be shared by several languages which are translated into the *CIL*. The *CTS* establishes a type safety, a faster execution of the code and provides a tool for easy cross-language integration. There is designed a shared library[9] which is used by languages based on *CIL*.

There are two main groups of types in *CTS* - *value types* and *reference types*. The variables of *value types* contain directly its value, on the other hand the variables of *reference type* point to a location of another value.

These two groups are reflected in C# via value semantic and referenced semantic[10] are mainly classes. Value semantic means that the value of variable is copied, when we use variable in an assignment or we pass it as a method argument. On the other hand the reference semantic means, that the reference on allocated memory is copied in case reference type variables.

The C# compiler does lots of work around value types (especially structs). Some optimalizations are performed and not all value copying in C# code could be seen in *CIL* code.

For instance, we have following C# code, where `MyStruct` is a struct:

[8] Common Type System

[9] The library is a part of ECMA 335 standard

[10] The *value types* are simple numeric types, structs and enumerations, whereas the *reference types* are classes, objects, arrays, strings. More details are in [7].

```
S1 MyStruct myStruct = new MyStruct();
S2 myStruct = new MyStruct(8);
```

We can conclude that there are three memory allocations and two value copyings. After translation into the *CIL*, there are not so many allocations or copyings. Produced *CIL* code looks as follows:

```
S3 ldloca.s 0
S4 initobj namespace.MyStruct
S5 ldloca.s 0
S6 ldc.i4.8
S7 call instance void valuetype
      namespace.MyStruct::.ctor(int32)
```

The statement S1 should allocate one memory place for struct, then copy it into the already allocated local variable named `myStruct`. This statement corresponds with *CIL* instructions in statements S3 and S4. The constructor in statement S1 is supplied by `initobj` instruction because it is parameterless. This instruction initalizes value on specified address with default values (`null` if the type is reference type, zero otherwise). The constructor in statement S2 is not parameterless, therefore it is supplied by `call` instruction. In the *CIL* code we work only with two allocated places and there is explicit copying. The copying is hidden in `initobj` instruction and in called constructor.

We also have to create a type system for ParallaX intermediate language. It is not necessarily required by any analyses, but type system helps to obtain more accurate information. We need a type system because the code will be transformed to C++ code finally and therefore the information about the types should be preserved.

It is important to distinguish the two main groups of data types, because they will be treated differently in C++. The variables with data type from the first group will be treated as a standard variable, but the variables of reference type will be treated as variables allocated on the heap and there are some pitfalls which must be solved.

3.9 Code Generation

The code generation is one of the final phases the ParallaX optimizer will be used for. It is important to take into consideration that there are some constructs which might be difficult or impossible to translate into target language. C# and

C++ are very similar languages, thus the set of that constructs will be very small.

As an example, consider the situation in Figure 3.2. There is used `dup` instruction and we have to solve how to propagate one value on two places. An easy way is to store the duplicated value into a temporary value. This also imposes that we will have to be able to add a new variable into the translated method. The question arises whether the `dup` instruction should have been removed during a transforming of *CIL* to the ParallaX intermediate language or during a code generation.

A more detailed code generation description could be found in a separate chapter.

3.10 ParallaX Intermediate Language Definition

In this section we will introduce the ParallaX intermediate language and its representation. We use the *PIL* in following chapters of this work to illustrate results of the analyses and transformations done by several algorithms mentioned in this work.

The *PIL* is mainly designed for easy use in ParallaX optimizer. It regards the requirements mentioned above. In this work the code will appear in three forms. The first form will consists of graphs, where each graph corresponds with *root instruction* defined bellow. The second one is formed by textual output. And the last one has form of an instruction list.

ParallaX intermediate language consists of *instructions* defined in Table 3.1. Any of these instructions can produce a value, which is then consumed by other instructions. All instructions can produce only one value, but the value might be consumed by more than one instruction. There is requirement that produced value must be consumed by at least one instruction.

Instructions with no produced values form a *method* of *PIL*, each of these we mark as *root instruction*. Instruction which is not a root one must produce a value, which is consumed by another instruction. We say that the first instruction is a *producer* of the second if the first one produces a value which the second instruction consumes. We also say that the second instruction is *consumer* of the first. We mark producers of an instrunction as its dependencies.

The instruction set of the *PIL* is smaller than the instruction set of *CIL*. The *CIL* instructions are aggregated into simpler ones. All instruction variants, which

```
1 LoadConst    30
2 Duplicate
3 Operation    Add
4 Store        loc1
```

Figure 3.4: First *PIL* representation illustrates the same program as Figure 3.3, i.e. the sum of two constant values and storing it into the variable `loc1`

differentiate between the precisions of the operands or has the operand encoded into its name, are ommited and supplied with only one instruction. For example the *CIL* instructions `ldloc.s`, `ldloc.1`, `ldarg.0`, `ldarg.s`, are supplied with the `Load` instruction. All the branch instructions of *CIL* are grouped into the instruction `Branch`.

All these modifications loses some information, i.e. operand precision, faster instructions etc. This information is no more needed, because the ParallaX *intermediate language* serves as a tool for generating of the code of high level programming language and we cannot take advantage of lost information in that language.

We will speak about the *CIL* code execution in the following paragraphs. We will consider that all root instructions are proccessed. If the instruction (root instruction or non-root instruction) is proccessed, all its value producers must be proccessed before. If the instruction produces a value, it is passed to its consument only once.

The first representation of *PIL* is the simplest one. It consists of a list of instructions, where each item is the name of the instruction and its operand (if it has operand). There is no difference between the *root instruction* and the normal instruction in the list. We can deduce that an instruction is the *root instruction* from the its type. However we cannot determine dependencies of an instruction, because it is not obvious which value producer from the list does it have, hence this form is not suitable for an illustration of relations between the instructions. The following representations of the *PIL* cover this drawback. As an example of this representation see the Figure 3.4.

The second representation consists of graphs. Each *root instruction* is expressed by a directed multi-graph $G = (V, E)$ where V is a set of instructions and E is a multi-set of relations between them. The edge corresponds to a value passing. The directed edge $\vec{e} = (u, v)$ denotes that instruction u produces the

Instruction	Produces	Consumes	Operand	Description
Nop	–	–	–	No operation
Load	loaded value	–	variable name	Loads value of variable
LoadConst	loaded constant	–	value	Loads numeric or string constant
Store	–	stored value	variable name	Stores value to variable
Operation	result of operation	operand 1, operand 2	operation type	Basic operation, list of operation is in Table 3.2
Branch	–	result of condition	jump target	Conditional or unconditional branch
New	address of array or object	[array size]	data type	Allocates new object
LoadAddress	address	–	variable name	Load address of some object
LoadIndirect	loaded value	address to load, array index	field name	Load value of variable on given address
StoreIndirect	–	address to store, array index, value	field name	Store value to variable on given address
Duplicate	value and value	value	–	Duplicate loaded value

Table 3.1: Instruction table of ParallaX *Intermediate Language*

value which the instruction v consumes. The graph is also *DAG* because there cannot be a cycle. This representation is familiar to the implementation in ParallaX optimizer. Each instruction has its own object. Their fields contains references to each other. Each reference refers to a value producer or consumer of the instruction. The implementation make the instruction manipulation more comfortable and transformation could be done without any obstructions. In this work we will show the graphs as a pictures.

The code snippet in the graph form might be too spacious, hence we will also use a textual form. In a textual form, the instructions are shown as a "simplified C code" in order to be short and brief. Each root instruction corresponds with a statement. `Duplicate` instruction which is displayed as an assignment to a temporary variable is the only exception. The instructions dependencies are visible from the syntax of the statement.

Operation	Description	Symbol
Nop	No operation	nop
Add	addition	+
Sub	substraction	-
Mul	multiplication	*
Div	division	/
Mod	module	%
And	bitwise and	&
Or	bitwise or	|
Xor	bitwise xor	^
Shl	bitwise left shift	<<
Shr	bitwise right shift	>>
Neg	negation	!
Not	bitwise not	~
Callvirt	virtual method call	callvirt
Call	method call	call
Ceq	equal to	==
Cne	non equal to	!=
Cgt	greater than	>
Cge	greater than or equal to	>=
Clt	less than	<
Cle	less than or equal	<=
True	true constant	1
False	false constant	0

Table 3.2: Table of operations, which might be used by *Operation* instruction

4. Control-Flow Analysis

There are two main parts of the original *control-flow*. The first part is the basic blocks detection and the building of a flowgraph. The second part is the detection of the loops or other control structures, i.e. `if`s, `goto`s, cycles - `for`s, `while`s. The control structures detection is done by the *interval analysis* and the structural analysis. The *interval analysis* identifies only loops, the *structural analysis* is more sophisticated and discovers more control structures (the list of the control structures is available in Section 4.3.1). In this chapter we will engage with a construction of a flowgraph (it includes a basic blocks detection) and with a *structural analysis*.

4.1 Problems

We can face problem with `duplicate` instruction. It may happen that the value produced by `duplicate` is consumed by two instructions each of them from different basic block. It can cause troubles when we change the control flow. In this case we can remove the `duplicate` instruction and supply it with the storing of the value to the local variable and loading it at appropriate places. The situation, when the value of the `duplicate` instruction is consumed in different basic block, is rare and common C# compilers[1] do not produce the *CIL* code which would end like this.

The algorithms of *structural analysis* presented in [10] also detect a region called *improper interval schema* (the exact definition of this term is written in Section 4.3). Some programming languages allow to create this situation, it is often caused by improper use of `goto`s. The only way how to produce that code in C# would be to use `goto` commands which would refer inside of nested blocks (cycle, if, try, etc.). However it is not possible, because the standard [7] forbids it. The `goto` command can refer to the label in the scope of the current block or in the scope of the outer block. The *CIL* code which is not produced by C# can contain the improper interval schema. In our situation we consider only the code generated from the C# code, hence we do not have to take the improper interval schema into account.

[1] The compilers provided in Microsoft Visual Studio or Mono Project.

We will use the same terms in the following paragraphs as in [10].

4.2 Basic blocks

As mentioned earlier *method* is formed by sequence of *root instructions* and their dependencies. Analyzed method will be considerably large, because of code inlining. We will group root instructions into *basic blocks*. We use the same definition of a basic block as in [1] and we use algorithm presented in [10].

We define *basic block* as a maximal sequence of consecutive root instructions which satisfy conditions:

- The only enter to the block is the first root instruction. Thus there is no jump into the middle of the basic block.
- The execution of the root instruction leaves the block on the last root instruction of the basic block. Hence there are no jumps or calls in the middle of the basic block.

Successors of basic block b are the basic blocks whose instructions could be executed immediately after the last root instruction of the basic block b. The predecessors of basic block b are the basic blocks which can be executed just before the first instruction of the basic block b.

We connect all the basic blocks where the method returns back with so-called terminating basic block. The reason is a comfortable use in other analyses. The points where the execution exits the method are unified with terminating basic block. The original basic block where the method returns back are still available via basic block connections.

The basic block of *PIL* code must begin at the first root instruction, at a root instruction, which is target of some branch, or a root instruction which immedieately follows conditional branch. These root instructions we will call *leaders*. In the algorithm for basic blocks construction we find *leaders* first. Then we create a basic block for each root instruction sequence which begins with a *leader* and ends with the root instruction before another *leader* or the last root instruction (the sequence does not contain any leader except the first root instruction). In the last phase we connect all basic blocks together according to branch instructions. The final algorithm for basic blocks contructions is shown in Algorithm 1.

```
   Data: The sequence instrs of PIL instructions
   Result: Flowgraph consisting of basic blocks
 1 Leaders = ∅;
 2 Leaders = Leaders ∪ first instruction of instrs;
 3 foreach branch instruction b in instr do
 4 |   if b is not the last instruction then
 5 |   |   Leaders = Leaders ∪ nextInstruction(b);
 6 |   end
 7 |   Leaders = Leaders ∪ branchTarget(b);
 8 end
 9 TerminatingBasicBlock = CreateTermBB();
10 BasicBlocks = {TerminatingBasicBlock};
11 foreach leader l in Leaders do
12 |   i = leader;
13 |   while i < length(instrs) ∧ i ∉ Leaders do
14 |   |   i = i + 1;
15 |   end
16 |   i = i − 1;
17 |   BasicBlocks = BasicBlock ∪ CreateBB(leader, i);
18 end
19 foreach basic block bb in BasicBlocks do
20 |   if bb is not TerminatingBasicBlock then
21 |   |   lastInstr = lastInstruction(bb);
22 |   |   if lastInstr is branch instruction then
23 |   |   |   if branchTarget(lastInstr) == −1 then
24 |   |   |   |   Succ(bb) = {TerminatingBasicBlock};
25 |   |   |   |   Prec(TerminatingBasicBlock) ∪= bb;
26 |   |   |   end
27 |   |   |   else
28 |   |   |   |   nextBB = basicBlockOf(branchTarget(lastrInstr));
29 |   |   |   |   Succ(bb) ∪= {nextBB};
30 |   |   |   |   Prec(nextBB) ∪= {bb};
31 |   |   |   end
32 |   |   |   if lastInstr has a value producer then
33 |   |   |   |   nextBB = basicBlockOf(lastrInstr + 1);
34 |   |   |   |   Succ(bb) ∪= {nextBB};
35 |   |   |   |   Prec(nextBB) ∪= {bb};
36 |   |   |   end
37 |   |   end
38 |   |   else
39 |   |   |   Succ(bb) = {TerminatingBasicBlock};
40 |   |   |   Prec(TerminatingBasicBlock) ∪= {bb};
41 |   |   end
42 |   end
43 end
44 return BasicBlocks;
```

Algorithm 1: Basic Block Construction

The graph $G = (V, E)$, where V is a set of basic blocks and E is a set of relations between the basic blocks, we call a flowgraph. A strongly connected component[2] of graph G we will call a *region*.

The flowgraph serves as an input for further analyses. In the following text we will apply the *structural analysis* to the flowgraph. It will extend the current flowgraph. We create a tree which will contain a control structures of the analyzed program. This tree is called *control tree*.

4.3 Structural Analysis

The *structural analysis* is kind of *control-flow* analysis. This analysis identifies control-flow structures, i.e. `if`s, `goto`s, cycles - `for`s, `while`s. This analysis gets a flowgraph of the analyzed method as an input. A *control tree* is its output. The set of nodes in the control tree consists *abstract nodes*. The abstract node represents a control structure consisting of other control structures which form theirs children. The *abstract nodes* in the leaves correspond with basic blocks.

4.3.1 Control Structures in C#

We will recognize several types of control structures. These control structures are language specific. The original C# control structures will be detected.

We may generate a C++ code without any information about the control structures, then the generated code would contain many `goto`s and it would not be suitable for the optimizer of the C++ compiler. Any other optimization needs to have control structures indentified.

We will detect following types of control structures: `Block`, `IfElse`, `For`, `While` and `Unknown`. Not all structures are covered by the first four types. We assign them to the last named group `Unknown`. For example the *switch* construct or *if* construct containing *goto* statement inside belongs to this group.

In the [10] there are other schemata recognized. Those are *proper* and *improper interval region*. Either group is connected with *goto*s statements.

The *proper interval region* is defined as an acyclic subgraph of a flowgraph, which does not match on the other recognized schemata. In C# context the *if* with *goto* statement inside match this definition. The *proper schema* is depicted

[2]The subgraph in which every vertex is reachable from the each other vertex

```
1  public static void ProperSchema(bool arg1,
2                                   bool arg2){
3      Console.WriteLine("B1");
4      if(arg1){
5          Console.WriteLine("B2");
6          goto B4;
7      }
8  B3:
9      Console.WriteLine("B3");
10     if(arg2){
11         Console.WriteLine("B5");
12         goto B6;
13     }
14 B4:
15     Console.WriteLine("B4");
16 B6:
17     Console.WriteLine("B6");
18 }
```

Figure 4.1: The C# source code of proper schema in Figure 4.2

in Figure 4.2. In the Figure 4.1 there is C# source code of this proper schema.

On the other hand the *improper interval region* is defined as a multi-entry region[3]. This pattern causes difficulties, because it could not be reduced easily. In programming language it could happen that the *goto* statement refers to the label inside a *loop*. This behaviour is in C# forbidden. The standard [7] says that *goto*s statements can point to the label inside the scope of the current or outer block, not the inner block. In C# there is only way how to create *improper region*. We have to create a cycle from `goto`s and one more entry to it. All other

[3]strongly connected component of the flowgraph

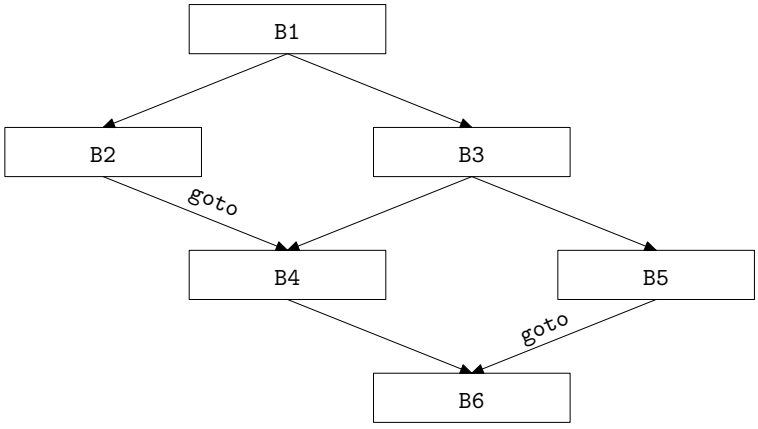


Figure 4.2: The proper schema

```
1  public static void ImproperSchema(bool arg1,
2
3      Console.WriteLine("B1");
4      if(arg1){
5          Console.WriteLine("B3");
6          goto B5;
7      }
8
9      Console.WriteLine("B2");
10 B4:
11     Console.WriteLine("B4");
12     if(arg2){
13         Console.WriteLine("B6");
14         goto B6;
15     }
16 B5:
17     Console.WriteLine("B5");
18     goto B4;
19 B6:
20     Console.WriteLine("B6");
21 }
```

Figure 4.3: The C# source code of improper schema in Figure 4.4

cycle structures in C# are considered to be a block, therefore we cannot create another entry to it. An example of *improper region* you can see in Figure 4.4. Its C# source code is in Figure 4.3.

In this work we will consider these two schemata to be in `unknown` construct group. If we do not use `gotos` statements, we do not match pattern of *proper* or *improper interval region*.

In the following sections we introduce the patterns which are recognized in the ParallaX optimizer. We also present the algorithms for their recognition. These algorithm can be included to the algorithm of structural analysis mentioned in [10]. We do not present the algorithm of structural analysis in this work, because it does not need any modifications.

Block Control Structure

The `Block` control structure is the simplest structure. It consists of several basic blocks which form a chain. The chain is formed by at least two basic blocks, it has entry and exit basic block. The other basic blocks forms a path from entry basic

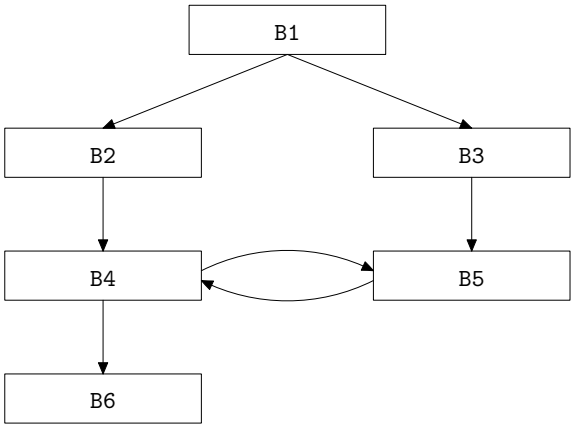


Figure 4.4: The improper schema containg a loop with two entries

block to exit basic block. In the algorithm we always get a `Block` with maximal length. The last basic block of the chain has to have only one successor in order to parse conditions of `IfElse` structure correctly. You can see an example of the `Block` control structure in Figure 4.5.

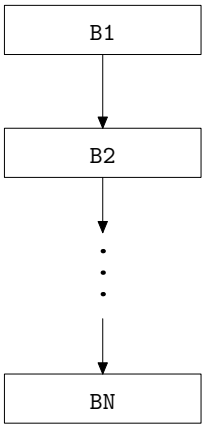


Figure 4.5: The `Block` control structure

IfElse Control Structure

Another more complex structure is `IfElse`. There are several parts of it: *condition*, *then* part and *else* part. The basic block which follows the control structure immediately is called *other* basic block. The control structure could contain the *condition* and *then* part more than once. The control structure is depicted on Figure 4.6. The conditions are in the dashed frames. The *else* part is not mandatory.

The *condition* part can be composed of many conditions, but at least one. Each condition except the first one must have only one predecessor. The first

one could have more than one successor. Each condition basic block also must have two successors. If the basic block has only one successor (which follows a condition basic block), it is *then*, *else* or *other* basic block. These basic block form a boundary of the control structure. To differentiate *other* basic block from *then* or *else* basic block we have to inspect its predecessor. If there is a such predecessor which has *other* basic block as its successor and it is only successor it means that the inspected basic block is *other* basic block.

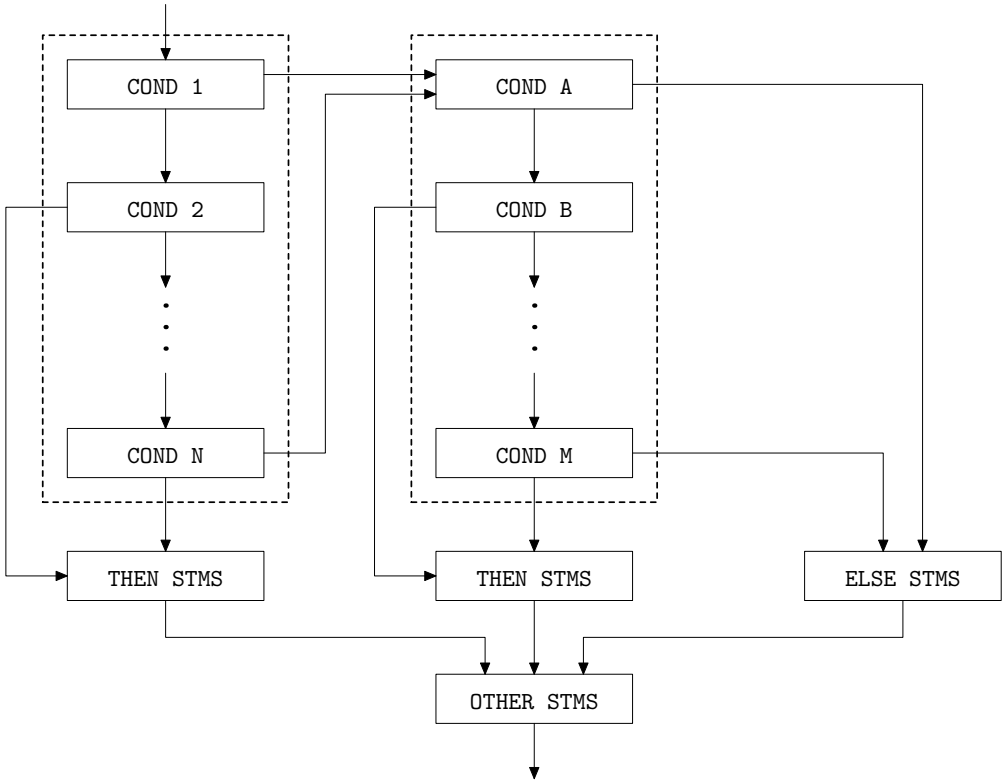


Figure 4.6: The `IfElse` control structure consisting of two *condition* parts (dashed frames) and one *else* part.

In the structural analysis algorithm (Algorithm **??**) we obtain a node and we test if it is an entry node of `IfElse` control structure. The detection is described in Algorithm 2. At first we search all *condition* blocks, *then* blocks and *other* block candidates. In the cited algorithm we use *depth first search*. Then we test if the size of each set of the blocks is correct and if all successors of the *then* blocks are the same *other* block (there must be only one *other* block).

For Control Structure

`For` control structure consists of several parts. It is illustrated in Figure 4.7. Same as in the previous control structure the `For` control structure contains a *for*

```
1  Function DetectIfElse(processedNode, out newNode, out nodeSet):
2      nodeSet=emptySet;
3      newNode=nil;
4      otherStructures = ∅;
5      thenStructures = ∅;
6      conditionStructures = ∅;
7      if |successors(processedNode)| == 2 then
8          visited = ∅;
9          DeepFirstSearch1(processedNode);
10         if |thenStructures| > 0 && |otherStructures| == 1 &&
           ∀t ∈ thenStructures : successors(t)[0] == otherStructures[0] then
11             nodeSet = conditionStructures ∪ thenStructures;
12             newNode = new IfElseStructure(nodeSet);
13             return;
14         end
15     end
16     return;
17 End
18 Function DeepFirstSearch1(abstractNode):
19     visited(abstractNode) = true;
20     if |predecessors(abstractNode)| == 1 &&
       ∃p ∈ predecessor(abstractNode) && |succesors(p)|==1 then
21         otherStructures ∪= abstractNode;
22         return;
23     end
24     else if |successors(abstractNode)|==1 then
25         thenStructures ∪= abstractNode;
26         return;
27     end
28     else
29         conditionStructures ∪= abstractNode;
30     end
31     foreach successor ∈ successors(abstractNode) do
32         if !visited(successor) then
33             DeepFirstSearch1(successor);
34         end
35     end
36 End
```

Algorithm 2: `IfElse` control structure detection

header part, a *condition* part, a *body* part and an *iteration* part. We also need to recognize a control structure following the `For` control structure; we denote it as *other* block.

For header part consists of only one block which is an entry point of the control structure. The only successor of this block is a *condition* block.

Condition part is composed of at least one condition block. Each partial condition block must have two successors and except first partial condition only one predecessor.

We cannot obtain any useful information from successors and predecessors of *other* block. The block could have one or more predecessors. At least one predecessor must be a condition block and the others could be another *condition* block or any *body* block. The `break` statement causes the connection between *body* block and *other* block.

The *body* part consists of at least one *body* block. We have to consider more than one block because it is not always possible to replace all the *body* blocks with a single block. The statements `break` and `continue` cause troubles, because of their connection to the control structure.

The *iteration* part is a block which follows the body and its only successor is the first *condition* block.

The Algorithm 3 identifies the *for header* at first, the *conditions* block are detected in the same way as in the case of *IfElse* control structure. The only pitfall is that some *body* block could satisfy the rules for a *condition* block identification, thus the algorithm must process the *condition* nodes again.

The blocks which follow the conditions (and are not the condition blocks) are either *other* block or *body* blocks. The algorithm inspects the path from them to the first *condition* block in order to differentiate them. If all paths from the inspected block to the first *condition* block contain a *for header block*, then the inspected node is the *other block*. If there is a path from the inspected block to the first *conditional* block which does not passes through the *for header* block, then it is a *body* block. All the blocks of the path from *body* block to the *iteration* block or *other* block are also *body* blocks.

Some of the *condition* blocks satisfy the conditions for identification of *condition* blocks. Those are the blocks which have only one predecessor and two successors. In Algorithm 3 the *condition* blocks are iterated in post-order (*depth-first search*) and if the one of the successors of the inspected block is the only

other block, then the inspected node is a *conditional* block and we stop searching. Otherwise the block is *body* block and the search must continue. This search must be done in the second stage because the *other* block was not available in the previous stage.

If the *body* part of the control construct consists of more than one *body* block, the structural analysis must be applied again. This time the input is the subgraph composed of *body* blocks and the edges which denotes that *break* or *continue* statement are deleted.

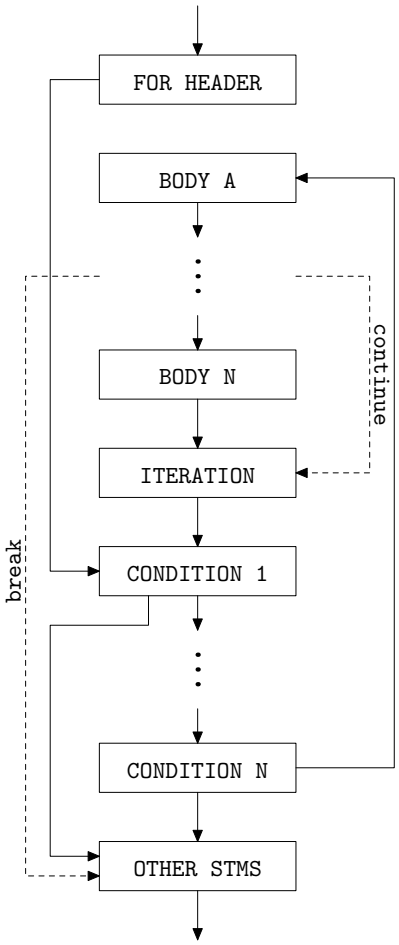


Figure 4.7: `For` control structure

While Control Structure

The `While` control structure is very similar to the `For` control structure. The `While` control structure is depicted in the Figure 4.8. The only difference is that the control structure misses the *for header* and *iteration* block. The *iteration* block could be supplied by the *first condition* block in Algorithm 3. The *for*

```
1  Function DetectIfElse(processedNode, allNodes, out newNode, out
   nodeSet):
2      nodeSet=emptySet;
3      newNode=nil;
4      forHeader = processedNode;
5      firstCondition = nil;
6      iteration = nil;
7      conditions = ∅;
8      bodyParts = ∅;
9      otherParts = ∅;
10     if |successors(processedNode)| == 1 then
11         processedNode = successors(processedNode)[0];
12         firstCondition = processedNode;
13         if predecessor(processedNode)[0] == forHeader then
14             iteration = predecessor(processedNode)[1];
15         end
16         else
17             iteration = predecessor(processedNode)[0];
18         end
19         visited = ∅;
20         DeepFirstSearch2(processedNode);
21         foreach condition ∈ conditions ;          /* in post order */
22         do
23             if !otherParts[0] ∈ successor(condition) then
24                 break;
25             end
26             conditions -= condition;
27             bodyParts ∪= condition;
28         end
29         bodyParts = Reduce(bodyParts);
30         if |bodyParts| == 1 && |conditions| > 0 && |otherParts| == 1
           && |predecessors(firstCondition)| == 2 &&
           |successsors(firstCondition)| == 2 then
31             nodeSet = forHeader ∪ conditions ∪ bodyParts;
32             newNode = new ForStructure(nodeSet);
33             return;
34         end
35     end
36     return;
37 End
```

Algorithm 3: `For` control structure detection

```
1  Function DeepFirstSearch2(abstractNode):
2      visited(abstractNode) = true;
3      if (|predecessors(abstractNode)| == 1 && |successors(abstractNode)|
       == 2) || abstractNode == firstCondition then
4          conditions ∪= abstractNode;
5          foreach successor ∈ successors(abstractNode) do
6              if !visited(successor) then
7                  DeepFirstSearch2(successor);
8              end
9          end
10     end
11     else if !Path(abstractNode, firstCondition, allNodes - forHeader) then
12         otherParts ∪= abstractNode;
13     end
14     else if Path(abstractNode, firstCondition, allNodes - forHeader) then
15         bodyParts∪= abstractNode;
16         foreach successor ∈ successors(abstractNode) do
17             if !visited(successor) then
18                 DeepFirstSearch2(successor);
19             end
20         end
21     end
22 End
```

Algorithm 4: Depth first search used in Algorithm 3

header block could be supplied by the predecessor of the first *condition* block which has the highest post-order number.

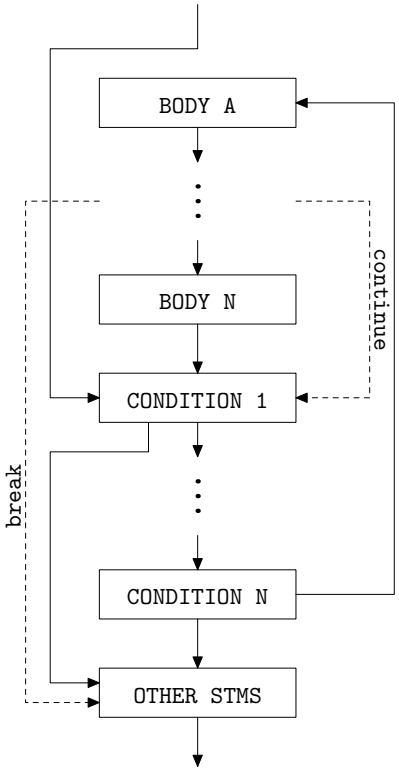


Figure 4.8: `While` control structure

Unknown Control Structure

The rest set of the block patterns is denoted as *unknown* control structures. We have to replace the minimal strongly connected component of the blocks with multiple entries and all blocks forming the paths from the nearest common *dominator*[4] of the entries to them. In similar way, we also replace the blocks which forms the paths from the exits of SCC to their nearest common *post-dominator*[5].

If there is no such SCC[6], we replace all blocks in the path from its nearest dominator to its nearest post-dominator.

[4]Dominators of a node are all nodes of the graph which are contained in all paths from the entry node of the graph to the node.

[5]Post-dominators of a node are all nodes of the graph which are contained in all paths from the node to the exit node.

[6]strongly connected component

4.3.2 Applicability to C#

The structural analysis algorithms are used for C# control structures recognition. The main algorithm remained unchanged, however the algorithms for the control structures recognition had to be adapted to C# context.

5. Points-to analysis

Points-to analysis also known as a *pointer analysis* produces the set of objects to which the variables of a pointer type may point during program execution. The points-to analysis give us same information as an *alias analysis*. We say that x is *alias* of y and y is *alias* of x if both x and y point to the same location. Alias analysis produces the pairs of variables which may be aliased at any moment of the runtime.

The points-to analysis differs from the alias analysis in the output. It does not produce pairs, but it makes a variable set for every pointer. The set contains objects to which the pointer may point. It requires less storage than the alias analysis. For instance for the assignments in C language `p = &x` and `p = &y` we get $p = \{x, y\}$, the alias analysis would produce pairs $(*p, x)$ and $(*p, y)$.

We distinguish two kinds of alias information - *may* alias information and *must* alias information. *May* alias information means that a variable *may* point to an object on any path in the flowgraph, on the other hand *must* alias information says that a variable *must* point to an object on all paths in the flowgraph.

There are two main algorithms of points-to analysis. The former which was presented in [3] was created by L. O. Andersen and the latter which was developed by B. Steensgaard and is described in [12].

Andersens' algorithm is more precise and runs in $\mathcal{O}(N^3)$ time. On the other hand Steensgaard's algorithm is not as precise as Andersens's is, but works in time $\mathcal{O}(N)$. We get sets from Steensgaard's algorithm. They are larger than those created by Andersen's algorithm and contain objects that are never pointed to by the pointer. See the example in section 5.4. In this work we shall mainly make use of Andersen's algorithm. We adapt it to the ParallaX intermediate language described in chapter 3 on page 10.

5.1 Accuracy

The analysis is called *flow-sensitive* if it takes the control-flow into consideration, otherwise it is *flow-insensitive*.

We make difference beetween information we received as summary for the whole function and that gained for a specific point (e.g. after each statement).

The former case is less precise and does not refer to the control-flow. The latter case is related to different points in the program and accuracy depends on the amount of observed points.

Many studies contain flow-insensitive points-to analysis because flow-sensitive analysis does not bring important information which would help to improve the code rapidly. One of the reasons was that the analysis was applied to small simple functions which make complex algorithms and points-to sets were processed by an interprocedural analysis.

In this work we have to deal with flow-sensitive points-to analysis, because we use only one function as an input. All called functions were inlined in previous phases of ParallaX. Processed function consists of basic blocks which then contain instructions. Basic blocks correspond with control structures like cycles and if construct. We use a basic block as a unit for creating point-to sets. That means that we will create sets of objects which pointers may point to after executing all statements (instructions) in basic block. This variant is at least as precise as flow-insensitive variant. However it is not fine-grained (it would not improve our results in comparison with small simple functions mentioned above).

In Andersen's work [3] there were also problems with calls. All the calls from the analysed function are considered to be different function calls even though they call the same function. If the recursion is detected, then the analysis uses a context of the first call of recursion as the context of a called method. It is where incaccuracy of the result starts increasing. it starts growing up inaccuracy of the result. This behaviour is in some aspects similar to function inlining, but there are some differences. For more details see [3].

Another interesting part of points-to analysis is accessing to fields of composed data types. There are three ways how to analyse fields. We have an expression `reference.SomeField` which describes access to a field named `SomeField` via a reference stored in the variable `reference`. The less precise way is called *field-indenpendent*. It ignores all fields and a points-to set of the field reference `SomeField` is the same as a points-to set of the variable `reference`. This approach mostly concerns in points-to analyses of *C code*. A more precise interpretation is called *field-based*. The points-to set of the reference and field are not identical, but the points-to set of the field is shared by many instances of the composed type, which the `reference` may point to. In the above mentioned expression the variable `reference` is ignored and the field `SomeField` is considered to be a single

variable. The most precise way is called *field-sensitive*. This interpretation differentiate between the instances of composed type, therefore all the fields of each instance (the reference points to) are treated as various variables. For instance let a `reference` may point to two memory places `A` and `B`. Then if we want to know the points-to set of `reference.SomeField` we get a union of the points-to sets `A.SomeField` and `B.SomeField` where `A` and `B` denote instances of the composed type. More details and examples could be found in the subsection **Field Load and Store**. The ParallaX optimizer implements the last mentioned variant, i.e. the *field-sensitive* interpretation.

5.2 *CIL* Intermezzo

There are two kinds of data types supported in *CIL* - *value types* and *reference types*. Variables of the value type directly contain their data, in contrast of variables of reference type which contain a reference to their data. The value types in C# are numeric ones (`int`, `float`, ...), structs and enums. A complete list can be found in [7] on pages 107-112. The reference types are classes, interfaces arrays or delegates. (see [7] on pages 112-117).

We could point to variables by pointers in *CIL*. There are *managed* and *unmanaged* pointers. Managed pointers may point to a local variable, parameter, field of a compound type, or an element of an array. They are registered by the *Garbage Collector*. In contrast to the managed pointers, unmanaged pointers are not registered by the *Garbage Collector*. We will not take the unmanaged pointers into consideration in this work. These pointers are not produced to *CIL* code from safe C# code.

In *CIL* code, which is generated by C# compiler, the instances of reference types are accessed via references that can be stored in the local variables, passed as arguments, be a field of an object or be an element of an array. We mostly use their addresses, because we access their fields or elements. In *CIL* there are several instructions used to access the reference type instances, for exapmle `ldfld` or `ldelem`. These instructions require the address of an object or an array. In *PIL* we will gather these instructions together in `LoadIndirect` and `StoreIndirect`.

Situation		Constraint	Meaning
Allocation	`New RefType` `Store locRef`	$locRef \lhd \{LOCREF\}$	$LOCREF \in pts(locRef)$
Assignment	`Load locRef` `Store copiedRef`	$copiedRef \lhd locRef$	$pts(copiedRef) \supseteq pts(locRef)$

Field Load	`LoadAddress locRef`
	`LoadIndirect RefType.Field`
	`Store copiedRef`
		$copiedRef \lhd locRef.Field$	$\forall LOC \in pts(locRef) :$
			$pts(copiedRef) \supseteq pts(LOC.Field)$

Field Store	`LoadAddress locRef`
	`Load srcRef`
	`StoreIndirect RefType.Field`
		$locRef.Field \lhd srcRef$	$\forall LOC \in pts(locRef) :$
			$pts(LOC.Field) \supseteq pts(srcRef)$

Table 5.1: Table with points-to constraints for various statements in ParallaX *Intermediate Language*

5.3 Algorithm Description

As mentioned above, the algorithm has two phases. At first constraints are generated and then a constraint-solving algorithm is applied. Local variables and arguments of analysed function having the reference type, will be regarded as a pointer variable. We also look on the variables with reference type representing a field of a composed type or an element of an array as a pointer variable. We will track the fields or elements of the instances which have been allocated inside the analysed function. We assume that all calls of the analysed function have been inlined and therefore there are not any `call` instructions.

5.3.1 Contraints Generating

There are several kinds of constraints. All the constraints are listed in Table 5.1. In the first phase we will try to extract these constraints from the *PIL* code. The constraint represents a direction of the change propagation of the altered points-to set. It is important to distinguish between the types of constraints, because the change propagation is different for each type of contraints.

We use capital letters and numbers to denote an *abstract location* (explained in subsection **Allocations** below). The name in camel case format beginning with a small letter referes to a variable name. For each variable v of the reference type we will keep a set of *abstract locations* and we will call it *points-to set* of variable v. We will denote it as $pts(v)$. Composed types, which are a kind of reference type, contain fields. Fields have their own name and type. To access that field we use a type or an *abstract location* of composed type and a field name. For example $ComposedType.FieldName$ or $LOC1.FieldName$.

Allocations

We associate each allocation in *PIL* (`New` instruction) with an *abstract location*. It represents the memory where the value is stored. The name of a variable which the allocated object is assigned to serve as a name for abstract location (we use it in capital letters in order to separate it from the variable name). Hence an allocation in the format `locVar = New()` has its name $LOCVAR$ (allocations in this format will be called *simple allocations*). However there are more complicated statements with `New` instruction like `locVar = New().Field` or a repeated allocation (these will be called *complex allocations*). We will provide these abstract locations with a numbered temporary name.

The detection of *abstract locations* is trivial, we look it up in DAG[1] of *root instruction*. *Simple allocations* can be recognized so that the instruction `New` is one of the value producers of the instructions `Store` or `StoreIndirect`, all the other occurences of the `New` instruction are *complex allocations*.

We create an apropriate constraint from Table 5.1 for all *simple allocation* It is not necessary to create any constraint for *complex allocation*, because the *abstract location* is not referenced

Simple allocation constraints $locRef \lhd \{LOCREF\}$ makes a subsequent impact on points-to set: $pts(locRef)$ must contain $LOCREF$ abstract location.

We shall also make difference if the reference is null or not, therefore we shall introduce special abstract location with name `NULL`. We also need to have an abstract location for that case when we do know nothing about the content of a variable - `UNKNOWN`. Such a situation comes, when variables are passed as arguments to the function. We know nothing about passed arguments without inter-procedural analysis and this analysis is not in the scope of this project.

On Figure 5.1 on line 6 we can see an allocation in C#. On Figure 5.2 on line 1-2 we can see the same code translated into *PIL*.

Assignments

Contraints of the assignment type are easy to recognize, because they must be in the format `copiedRef = locRef` where `copiedRef` and `locRef` are local variables or the passed arguments. Other assignments with dereferenced variables are described in the **Field Load and Store** subsection.

[1] Directed Acyclic Graph

```
1 class ReferenceType
2 {
3     public ReferenceType Field;
4 }
5
6 ReferenceType reference = new ReferenceType();
7 ReferenceType copiedReference = reference;
8 copiedReference = reference.Field;
9 reference.Field = copiedReference;
```

Figure 5.1: The situations in C# code which affects the point-to sets. Line 6 contains a new object creation, the assignment of a variable on line 7, loading fields of an object on line 8 and storing them on line 9

```
1  New             ReferenceType
2  Store           locReference
3
4  Load            locReference
5  Store           copiedReference
6
7  LoadAddress     locReference
8  LoadIndirect    ReferenceType.Field
9  Store           copiedReference
10
11 LoadAddress     locReference
12 Load            copiedReference
13 StoreIndirect   ReferenceType.Field
```

Figure 5.2: The situations in ParallaX *Intermediate Language* which affects point-to sets. Lines 1-2 contains a new object creation, the assignment of a variable on lines 4-5, loading fields of an object on lines 7-9 and finally storing them on line 11-13

We have the constraint $copiedRef \lhd locRef$ demanding, that $copiedRef$ may point to the same abstract location as $locRef$ may, i.e. $pts(copiedRef) \supseteq pts(locRef)$.

Field Load and Store

The loading and storing of the fields is the most complicated situation. In Table 5.1 there are only cases mentioned with one dereference on the left or the right side of an assignment. The assignment of more than just one dereference could be decomposed into more statements on condition that each statement has at most one dereference (i.e. it contains only one instruction `StoreIndirect` or `LoadIndirect`).

We need temporary variables, so as to decompose a complex statement. We replace all derefences inside another one by a temporary variable. Consider the following statement as an example of decomposing:

```
loc1.Field1.Field2 = loc2.Field3.Field4
```

Results of the previous statement decomposition:

```
t1 = loc1.Field1
t2 = loc2.Field3
t3 = t2.Field4
t1.Field2 = t3
```

At first we analyse the field store situation. Let us have a statement in the format *expr1.CustomField = expr2* where *expr1* and *expr2* can be variables or field variables.

If *expr1* is a field variable *anotherExpr.AnotherField* we create a new temporary variable *t1* and try to analyse a new expression *t1 = anotherExpr.AnotherField*. If *expr2* is also an expression we replace it by *t2* in the same way and create constraint $t1.CustomField \lhd t2$.

If we have a statement in the format *variable = expr.CustomField* where *expr* is a field variable, then we substitute it by a temporary variable t again and create a new constraint *variable* $\lhd$ *t.CustomField*. In the end we also have to analyze a new expression $t = expr$.

The last possible expression *variable1 = variable2.CustomField* generates a constraint *variable1* $\lhd$ *variable2.CustomField*.

When we have decomposed all complex field loads and stores, we obtain constraints in the format $variable1.CustomField = variable2$ for a field store and another one in the format $variable1 = variable2.CustomField$ for a field load. For the first constraint there must be the following equation satisfied:

$$\forall\ LOC \in pts(variable1) : pts(LOC.CustomField) \supseteq pts(variable2)$$

and for the second constraint:

$$\forall\ LOC \in pts(variable2) : pts(variable1) \supseteq pts(LOC.CustomField)$$

Notice that we keep points-to set for a field variable of an instance of the composed type. The name of its *abstract location* serves as a signature of the instance. We also define the points-to set of the dereferenced variable as follows:

$$pts(variable.CustomField) = \bigcup_{LOC \in pts(variable)} pts(LOC.CustomField)$$

This is not used while the algorithm propagates the changes, but it is included in the result, because it provides more accurate information about the field usage for us.

5.3.2 Points-to Sets Propagation

During first phase of the points-to analysis we have generated a set of constraints, which we use to propagate points-to sets among variables now. These constraints are an input for the propagating algorithm. A points-to set for each variable is an output of the algorithm.

The propagating algorithm is depicted in Algorithm 5. It uses a worklist where the variables, whose points-to set has changed, are stored.

The outer `while` is neccessary for the correctness of the algorithm. For example let variables `loc1` and `loc2` point to the same abstract location $\{LOC1\}$. They are both aliases for each other. A statement changes `loc1.Field` and another reads `loc2.Field`. We cannot rely on `loc2` getting into the worklist.

Lines 26-38 are required for correctness. On the other hand lines 14-24 are not necessary, but empirical observations have prooved that this heuristic speeds

up the algorithm.

5.4 Steensgaard's Algorithm

Steensgaard's algorithm does not use the subset based constraints, but it uses a type inference. It means that an artificial type is made for each variable and a constraint for each statement of the program. Constraints make relations between the types. The types are unified according to a rule if there is statement `x = y`, then $type(x) = type(y)$. The points-to sets are associated rather with the type than the variable itself in this algorithm, therefore if $type(x) = type(y)$ then $pts(x) = pts(y)$ has to be satisfied. In addition each type can point to at most one other type.

For example, consider the situation on Figure 5.3a, there are pointers pointing to each other. The frame surrounding each variable indicates its type and its points-to set[2]. The situation after processing statement `x = y` is depicted on Figure 5.3b. In that point both `x` and `y` may point to the same variable `w`. The variable types of `x` and `y` should be unified, because of the condition mentioned above, this is illustrated on Figure 5.3c. The type of `u` and `w` points to two different types and this is forbidden, thus we have to unify these two types, see Figure 5.3d.

The result of the analysis shown on Figure 5.3d gives us information that both `u` and `w` may point to `v` or `z`. We have obtained inaccurate information, because in this context it is obvious that `u` cannot point to `z` and also `w` could not point to `v`. This redundant information has an impact only on precision of depending analyses, the correctness should not be threatened, because we have *may* information not *must* information.

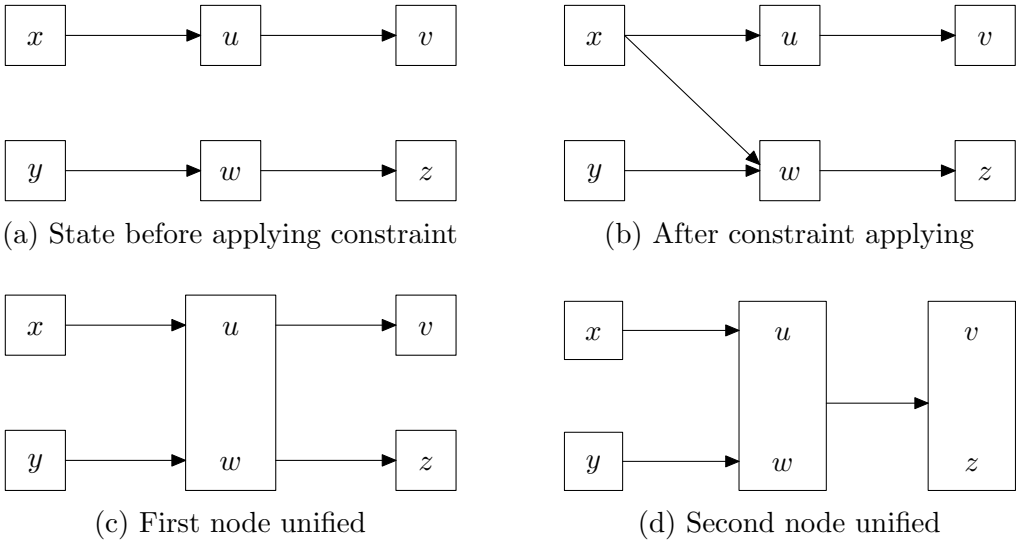
5.5 Usage

Another code analysis requires the result of this analysis. *Depependence analysis* is the cited analysis. In context of C# and C++, it can be used to detect the validity range of a variable. The *Garbage Collector* is responsible for the allocation and the deallocation in C#. There is no *Garbage Collector* in C++, therefore we

[2]points-to se is associated with the type

```
   Data: The set of constraints
   Result: Points-to sets for each variables
 1 Initialize pts(v) as empty set for each variable v except function arguments;
 2 Set pts(a) = {UNKNOWN} for each function argument a;
 3 foreach constraint v ◁ {LOC} do
 4 |   pts(v) = pts(v) ∪ {LOC};
 5 end
 6 W = {v | pts(v) ≠ ∅};
 7 while W is not empty do
 8 |   while W is not empty do
 9 |   |   w = get from W;
10 |   |   foreach assignment constraint u ◁ w do
11 |   |   |   pts(u) = pts(u) ∪ pts(w);
12 |   |   |   Add u to working list W;
13 |   |   end
14 |   |   foreach field store constraint u.CustomField ◁ v where u = w or
   |   |   v = w do
15 |   |   |   foreach LOC ∈ pts(u) do
16 |   |   |   |   pts(LOC.CustomField) = pts(LOC.CustomField) ∪ pts(v);
17 |   |   |   end
18 |   |   end
19 |   |   foreach field load constraint u ◁ w.CustomField do
20 |   |   |   foreach LOC ∈ pts(w) do
21 |   |   |   |   pts(u) = pts(u) ∪ pts(LOC.CustomField);
22 |   |   |   |   Add u to working list W;
23 |   |   |   end
24 |   |   end
25 |   end
26 |   foreach field store constraint u.CustomField ◁ v do
27 |   |   foreach LOC ∈ pts(u) do
28 |   |   |   pts(LOC.CustomField) = pts(LOC.CustomField) ∪ pts(v)
29 |   |   end
30 |   end
31 |   foreach field load constraint u ◁ v.CustomField do
32 |   |   foreach LOC ∈ pts(v) do
33 |   |   |   pts(u) = pts(u) ∪ pts(LOC.CustomField)
34 |   |   end
35 |   |   if pts(u) has changed then
36 |   |   |   Add u to working list W;
37 |   |   end
38 |   end
39 end
40 foreach variable v do
41 |   if {UNKNOWN} ∈ pts(v) then
42 |   |   pts(v) = {UNKNOWN};
43 |   end
44 end
```

Algorithm 5: Worklist propagation



(a) State before applying constraint

(b) After constraint applying

(c) First node unified

(d) Second node unified

Figure 5.3: Applying constraint for `x = y` statement in the Steensgaard's algorithm

have to manage the allocations and deallocations ourselves. Information provided by *Points-to analysis* helps us to determine the validity range of the variable.

6. Dependence Analysis

Dependence graphs are connected with *Dependence analysis* which is an important tool for the instruction scheduling. This work is dealing with C# code transformed up to intermediate language, where the instruction scheduling is not as important as in a native code. We use the information provided by analysis for a revealing the parts of program which may run in parallel.

As main parts of analysis are generetad data structures which contain information about dependencies in analyzed program. There are two kinds dependencies which we study, the former *data dependences* and the latter *control dependences*. The mentioned datastructures are called *data-dependence graph* and *control-dependence graph*. Both of them are *program-dependence graph*.

In following sections we examine data dependences and control dependences in more details. All analyses will be adapted to context of C#.

6.1 Data dependences

Data dependence is a constraint which reflects a relation between two statements. These two statements use the same memory location. There are several kinds of *data dependences*. They are contained in many publications. We use the definitions from [10].

We assume that a statement `S1` precedes a statement `S2`. Then we consider following categories of *data dependences*:

- If the former statement `S1` sets a value that the latter statement `S2` uses, we call this a *flow dependence* or *true dependence*.
- If `S1` uses the value which the `S2` sets, we say that there is an *antidepence* between them.
- If both statements `S1` and `S2` write to the same variable, we call this depencence an *output dependence*.
- Finally, if `S1` and `S2` read the same variable's value, we say that there is an *input dependence* between them.

Data dependence graph is created from *data dependences*. If we consider the statements only inside a basic block, then the graph is DAG[1]. The dependences inside the basic block is useful when the instructions are scheduled to fully utilize processor and memory. We do not concern the instruction scheduling, because we use *PIL* and it is not a final form of the code.

The *data dependences* are important for ParallaX development environment, because the automatic parallelization on the code must not break the *data depences* in order to preserve program correctness. Important dependences are those between the basic blocks and loop iterations, because that is a possible area for automatic parallelization which could not be affected by C++ compiler rapidly in later stages of the code processing.

There are many methods how to inspect the *data dependences*. The technics revealing the dependences between the statements which operates with the array variables are called *dependence testing*. There are many algorithms which recognize only specific types of the *data dependences*. For example:

- Acyclic test
- Fourier Motzkin test
- Omega test
- Simple Loop Residue test
- Extended GCD test
- ... and more others

The domain of *data dependences* is too wide and needs detailed inspection of this area to mark out the technics which can be used in the context of the ParallaX development environment, therefore we will study only *control dependences* in more detail in this work.

6.2 Control dependences

At first we introduce a relation *dominance* on flowgraph nodes (this definition is presented in [10]). We say that basic block d *dominates* basic block i, written

[1]Directed Acyclic Graph

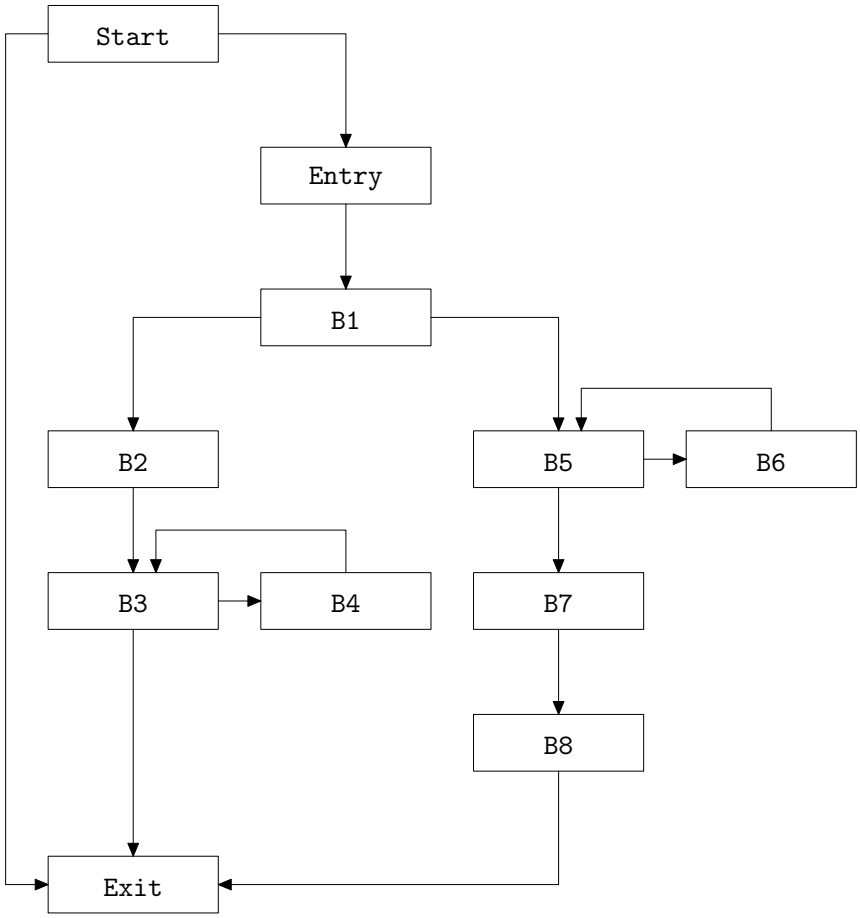


Figure 6.1: Example of the extended flowgraph

d dom i, if every possible execution path from `entry` to i includes d. It is clear that this relation is reflexive, transitive and antisymmetric. In the similar way we define *postdominance* relation. The only difference is in the path. We say that basic block p *postdominates* basic block i, written p *pdom* i, if every possible execution path from i to `exit` passes through p.

We use the previous definitions to define the *control dependence*. Basic block n is *control-dependent* on basic block m if and only if

1. there exists a path in the flow graph from m to n. All basic blocks in that path except the m are postdominated by n and

2. n does not postdominates m.

The *control dependency* relation could be expressed as a *control dependence graph*. We use the extended flowgraph to build the CDG. The extended flowgraph consists of the original one with added one node called *start*. This node is connected to the entry node and the exit node. This steps preserves the output CDG as one connected component.

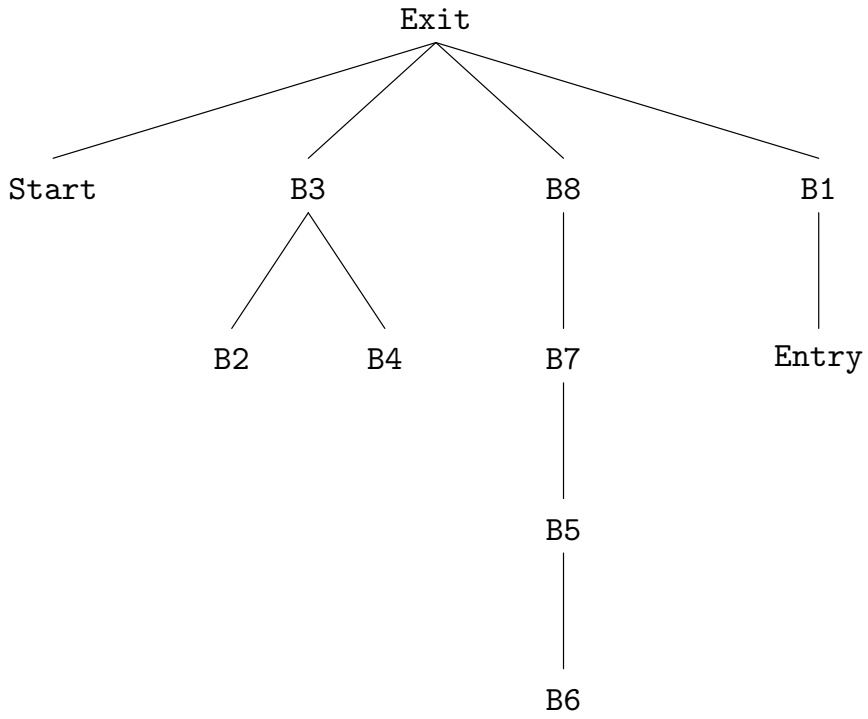


Figure 6.2: Postdominance tree of the flowgraph in Figure 6.1

We start with constructing postdominance tree[2]. We create a set of edges S which consists of the edges $m \to n$ in the extended flowgraph which satisfies the condition that n does not postdominates m. Next, we find the nearest common ancestor a of these nodes in the postdominance tree. All the nodes on the path from a to n except a are control dependent on m.

Let consider a flowgraph in Figure 6.1. We build a post dominance tree which is depicted in Figure 6.2. In the next step we create the set S. It contains following edges: `B3`→`B4`, `B1`→`B2`, `B1`→`B5`, `B5`→`B` and `Start`→`Entry`. The Figure 6.3 illustrates the final *control dependence graph*.

6.3 Conditions for Parallel Running

The *control dependence graph* provides us information about parallel execution of the basic blocks. If the node a and b are control-dependent on the same node c and there is no data dependence between them, then they can be executed in parallel. For instance in Figure 6.3 the basic blocks `B3` and `B2` are control-dependant on `B1`,hence they can run in parallel if there is no data-dependence.

[2]Graph which is built according to postdominance relation.

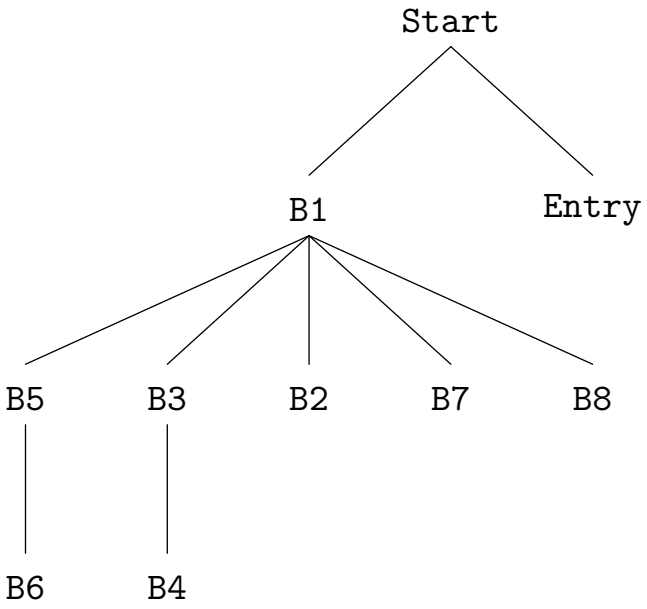


Figure 6.3: The control dependence graph of the flowgraph in Figure 6.1

6.4 Use in C# context

The algorithm which builds the CDG operates with a flowgraph which consists of the basic blocks. These basic blocks are formed by *PIL* instructions. The output is a new graph composed of the same basic blocks again. Therefore no changes are needed and well known algorithms (mentioned in [10]) could be used.

7. Conclusion

In this work we have designed the ParallaX intermediate language[1], which was not defined as a main goal of this work. It has turned out that the *PIL* is an essential part of the ParallaX environment. The algorithms of the *structural*, *points-to* and *dependence* analysis were adapted to use *PIL*.

The *structural analysis* gets a sequence of *PIL* instructions as its input. It produces two basic graphs - *control graph* and *flow graph*. There are several C# control structures detected (if conditions, for cycles etc.). The other control structures can exist in a valid *CIL* or *PIL* code. However they cannot be produced by the common C# compiler[2].

There are two main approaches of doing the *points to* analysis. The first one is less accurate Steensgaard's method described in [12]. The second one is more accurate Andersen's method introduced in [3]. We have chosen Andersen's method. Both versions analyse the *C* code but not the C# code, therefore the algorithms had to be modified appropriately. Either version does not focus on the complex data types and produces less precise information about them. We have focused on the analysis of class fields in this work. The analysis was motivated by the existing analysis for Java described in [8]. The implemented analysis gets a method in *PIL* as an input. The analysis produces sets of *abstract locations* for all variables of the reference type. The *abstract locations* correspond with instructions which allocate a new memory.

There are two main types of the dependences which have been studied in the scope of the *dependence analysis*. Those are the *control dependences* and the *data dependences*. This work only includes the control dependences. The area of the *data dependeces* has appeared to be too wide and needs to be examined in more detail in order to be used in the context of the ParallaX environment.

The *control dependence* analysis requires a produced flowgraph, which has been provided by the *structural analysis*. The *post dominance tree* is also used to detect control dependences.

You can find the source codes of the implemented analyses on the attached DVD. It is not possible to test the entire ParallaX development environment, because not all parts has been implemented yet. More details about testing of

[1] *PIL*

[2] The compilers provided in Microsoft Visual Studio or Mono Project.

analyses results are placed into the file with name `readme.txt`. You can find this file in the root of DVD filesystem.

This master thesis has reached the defined goals. The main contributions consist in exploring of the existing algorithms of the selected analyses and adapting them to the context of the C# programs. The implementation has become a part of the larger system - the ParallaX environment, which prepares and transforms programs for the parallel execution. It also provides a solid basis for further work in the context of the ParallaX environment.

Bibliography

[1] A.V. Aho, M.S. Lam, J.D. Ullman, and R. Sethi. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2011.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2002.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[4] Mike Barnett, Manuel Fähndrich, Diego Garbervetsky, and Francesco Logozzo. Annotations for (more) precise points-to analysis. In *Proceedings of the 2nd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO'07)*, pages 11–18, 2007.

[5] David Bednárek, Jiří Dokulil, Jakub Yaghob, and F Zavoral. The bobox project parallelization framework and server for data processing. *Charles University in Prague, Technical Report*, 1:2011, 2011.

[6] Michal Brabec. Parallelizability analysis based on bytecode. Master's thesis, Charles University in Prague Faculty of Mathematics and Physics, 2013.

[7] ECMA. *Standard ECMA-334 C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 2006.

[8] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using spark. *Compiler Construction*, pages 153–169, 2003.

[9] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. *In Proceedings of PASTE'01*, pages 73–79, 2001.

[10] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[11] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL '97, pages 1–14, New York, NY, USA, 1997. ACM.

[12] Bjarne Steensgaard. Points-to analysis in almost linear time. *Proceeding POPL '96 Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages History of programming languages*, pages 32–40, 1996.

[13] Linda Wilkens. The joy of teaching with C#. *Journal of Computing Sciences in Colleges*, Volume 19 Issue 2:254–264, 2003.

List of Tables

List of Figures

List of Abbreviations

CIL Common Intermediate Language

CLR Common Language Runtime

CTS Common Type System

CDG Control Dependences Graph

DAG Directed Acyclic Graph

JIT Just In Time Compiler

PIL ParallaXIntermediate Language

STL Standard Template Library

A. Outputs

Following pages contain a demonstration of the points-to analysis. These outputs could be obtained from the implemented analysis by running the tests. More informations are contained in the attached `readme.txt` file.

The first code snippet is always analyzed C# code. The second snippet is a translation of C# code to the ParallaX *intermediate code*. Finally the last output are points-to set for the code in *intermediate language*.

A.1 Common structure

```
1  public class SimpleClass
2  {
3      public bool boolValue;
4      public int integralValue;
5      public long longValue;
6      public double doubleValue;
7      public float floatValue;
8      public SimpleClass selfReference;
9
10     public int Field { get; set; }
11
12     public int IntegralValue
13     {
14         get { return this.integralValue; }
15         set { this.integralValue = value; }
16     }
17
18     public bool Method()
19     {
20         return true;
21     }
22 }
```

A.2 Example 1

C# code:

```
1  void Test1(){
2      SimpleClass sc1 = new SimpleClass();
3      SimpleClass sc2 = new SimpleClass();
4      sc2.selfReference = sc2;
5      sc2.selfReference = sc1;
6
7      SimpleClass sc3 =
          PointsToTest.SetIntegralValue(new
          SimpleClass());
8      SimpleClass sc5 = sc3;
9      sc5.selfReference = sc2;
10
11     SimpleClass sc4 =
          sc3.selfReference.selfReference;
12     sc3.selfReference.selfReference = sc3;
13
14     int iterations = 5;
15     while (iterations-- > 0)
16     {
17         SimpleClass scWhile = new SimpleClass();
18         scWhile.selfReference = sc1;
19         sc1 = scWhile;
20     }
21 }
```

Intermediate language code:

```
1  Nop
2  Loc0 ← New()
3  Loc1 ← New()
4  *(Loc1).selfReference ← Loc1
5  *(Loc1).selfReference ← Loc0
6  Loc8 ← New()
7  *(Loc8).integralValue ← 5
8  Loc9 ← Loc8
9  GoTo 9
10 Loc10 ← Loc9
11 GoTo 11
12 Loc2 ← Loc10
13 Loc3 ← Loc2
14 *(Loc3).selfReference ← Loc1
15 Loc4 ← *(*(Loc2).selfReference).selfReference
16 *(*(Loc2).selfReference).selfReference ← Loc2
17 Loc5 ← 5
18 GoTo 21
19 Loc6 ← New()
```

```
20 *(Loc6).selfReference ←  Loc0
21 Loc0 ←  Loc6
22 Loc5 ←  OpSub(T0 ,1)
23 Loc7 ←  OpCgt(T0 ,0)
24 if Loc7 then GoTo 18
25 Return
```

Points-to sets:

```
Loc0 = {LOC0, LOC6}
Loc1 = {LOC1}
Loc8 = {LOC8}
Loc6 = {LOC6}
Loc9 = {LOC8}
Loc10 = {LOC8}
Loc2 = {LOC8}
Loc3 = {LOC8}
Loc4 = {LOC1, LOC0, LOC6, LOC8}
Loc1::selfReference = {LOC1, LOC0, LOC6, LOC8}
Loc3::selfReference = {LOC1}
Loc2::selfReference = {LOC1}
Loc2::selfReference::selfReference = {LOC1, LOC0,
   LOC6, LOC8}
Loc6::selfReference = {LOC0, LOC6}
```

A.3 Example 2

C# code:

```
1 void Test2(){
2     SimpleClass sc1 = new SimpleClass();
3     SimpleClass sc2 = new SimpleClass();
4
5     SimpleClass sc3 = sc2;
6     sc3.selfReference = sc1;
7     sc1 = sc2.selfReference;
8 }
```

Intermediate language code:

```
1 Nop
2 Loc0 ← New()
3 Loc1 ← New()
4 Loc2 ← Loc1
5 *(Loc2).selfReference ← Loc0
6 Loc0 ← *(Loc1).selfReference
7 Return
```

Points-to sets:

```
Loc0 = {LOC0}
Loc1 = {LOC1}
Loc2 = {LOC1}
Loc2::selfReference = {LOC0}
Loc1::selfReference = {LOC0}
```

A.4 Example 3

C# code:

```
1  void Test3{
2      SimpleClass sc1 = new SimpleClass();
3      SimpleClass sc2 = new SimpleClass();
4      sc2.selfReference = new SimpleClass();
5      sc2.selfReference.
6          selfReference = new SimpleClass();
7      sc2.selfReference.
8          selfReference.
9          selfReference = new SimpleClass();
10     sc2.selfReference
11        .selfReference
12        .selfReference
13        .selfReference = sc2;
14
15     sc1 = sc2.selfReference.selfReference;
16 }
```

Intermediate language code:

```
1  Nop
2  Loc0 ← New()
3  Loc1 ← New()
4  *(Loc1).selfReference ← New()
5  *(*(Loc1).selfReference).selfReference ← New()
6  *(*(*(Loc1).selfReference).selfReference)
7           .selfReference ← New()
8  *(*(*(*(Loc1).selfReference).selfReference)
9           .selfReference).selfReference ← Loc1
10 Loc0 ← *(*(Loc1).selfReference).selfReference
11 Return
```

Points-to sets:

```
Loc0 = {LOC0, LOC1::SELFREFERENCE::SELFREFERENCE}
Loc1 = {LOC1}
Loc1::selfReference = {LOC1::SELFREFERENCE}
Loc1::selfReference::selfReference =
        {LOC1::SELFREFERENCE::SELFREFERENCE}
Loc1::selfReference::selfReference::selfReference =
        {LOC1::SELFREFERENCE::SELFREFERENCE::SELFREFERENCE}
Loc1::selfReference::selfReference::selfReference
        ::selfReference = {LOC1}
```

A.5 Example 4

C# code:

```
1  public void Test4()
2  {
3      SimpleClass sc1 = new SimpleClass();
4      SimpleClass sc2 = new SimpleClass();
5      sc1 = new SimpleClass();
6      SimpleClass sc3 = sc1;
7
8      SimpleClass sc4 = new SimpleClass();
9      SimpleClass sc5 = sc4;
10     sc5 = new SimpleClass();
11     sc5.selfReference = new SimpleClass();
12     SimpleClass sc6 = sc4.selfReference;
13 }
```

Intermediate language code:

```
1  Nop
2  Loc0 ← New()
3  Loc1 ← New()
4  Loc0 ← New()
5  Loc2 ← Loc0
6  Loc3 ← New()
7  Loc4 ← Loc3
8  Loc4 ← New()
9  *(Loc4).selfReference ← New()
10 Loc5 ← *(Loc3).selfReference
11 Return
```

Points-to sets:

```
Loc0 = {LOC0, TEMP1}
Loc1 = {LOC1}
Loc3 = {LOC3}
Loc4 = {LOC4, LOC3}
Loc2 = {LOC0, TEMP1}
Loc5 = {LOC4::SELFREFERENCE}
Loc4::selfReference = {LOC4::SELFREFERENCE}
Loc3::selfReference = {LOC4::SELFREFERENCE}
```

B. DVD Content

For the detailed instructions to run the project see `README.txt` file. The attached DVD has following structure:

- parallax
  - trunk
    - Architecture
    - Documentation
    - Mono.Cecil
    - ParallaX
    - ParallaX.Common
    - ParallaX.ConfigurationEngine
    - ParallaX.Core.Analyzer
    - ParallaX.Core.CodeGenerator
    - ParallaX.Core.CodePreprocessor
    - ParallaX.Core.DependenceTester
    - ParallaX.Core.PrelimCodeAnalyzer
    - ParallaX.Core.PrelimTransformer
    - Parallax.Interface
    - ParallaX.IntermediateLanguage
    - ParallaX.Library
    - ParallaX.OptimalizationDriver
    - UnitTest
    - Local.testsettings
    - ParallaX.sln
    - ParallaX_Example1.sln
    - ParallaX.vsmdi
    - Settings.StyleCop
    - TraceAndTestImpact.testsettings
- parallax_example

  - trunk
    - ParallaXExample1
    - ParallaXExample1.Lib
    - ParallaxExample2
    - common.bat
    - ParallaXExample1.sln
- README.txt