Report

The correctness of the definite assignment analysis in C#

Author(s):
Fruja, Nicu Georgian

Publication Date:
2004

Permanent Link:
https://doi.org/10.3929/ethz-a-006733196

Rights / License:
In Copyright - Non-Commercial Use Permitted
The correctness of the
definite assignment analysis in C♯

Nicu G. Fruja
Computer Science Department, ETH Zürich, CH–8092 Zürich, Switzerland
fruja@inf.ethz.ch

Abstract

The C♯ compiler must carry out a specific conservative flow analysis to ensure that every local variable is definitely assigned when any access to its value occurs. This prevents access to uninitialized memory and is a crucial ingredient for the type safety of C♯. We formalize the rules of the definite assignment analysis of the C♯ compiler with data flow equations and we prove the correctness of the analysis, i.e. if the analysis will infer a local variable as definitely assigned at certain program point, then this variable will actually be assigned at that point during every execution of the program.

1 Introduction

Let us suppose that an attacker wants to fool the C♯ type system. His idea is expressed by the next block:

```csharp
{ int[] a;
  try {a = (int[])(new object());}
  catch(InvalidCastException)
  {Console.WriteLine(a[7]);}
}
```

A pure `object` is type casted into an array of integers. The attacker thinks the following will work: after the `InvalidCastException` which is thrown at runtime is caught, the `object` can be used in the handler of the `catch` clause, as an array to generate unpredictable behavior. Some people might think that a `NullReferenceException` is thrown at runtime when `a[7]` is accessed and thus the attacker will not succeed. Actually, his idea does not work since the block is rejected already at compile time due to the definite assignment analysis. Through this analysis, the C♯ compiler infers that `a` might not be assigned in one execution path to the access of `a[7]`. The analysis states that `a` is not definitely assigned at the beginning of the `catch` block since it is not definitely assigned at the beginning of the `try` statement.

A necessary condition for C♯ to be a type safe language is the following: whenever an expression is evaluated, the resulting value is of the type of the expression. If we suppose that a local variable is uninitialized when its value is required, the execution proceeds with the arbitrary value which was at the memory position of the uninitialized local variable. Since this value could be of any type, we would obviously violate the type safety of C♯ and we could easily produce unpredictable behavior.

Since local variables are not initialized with default values like for example static variables or instance variables of class instances, a C♯ compiler must carry out a specific conservative flow analysis to ensure that every local variable is definitely assigned when any access to its value occurs. This definite assignment analysis which is a static analysis (see §5.3 for other static analyses) has to guarantee that there is an initialization to a local variable on every possible execution path before the variable is read. Since the problem is undecidable in general, the C♯ Language Specification §5.3 contains a definition of a decidable subclass. So far, the definite assignment analysis of the Java compiler has been formalized with data flow equations in the work of Stärk et al. [11] and related to the problem of generating verifiable bytecode from legal Java source code programs. A formalization of the analysis for Java which uses type systems is presented in [9]. Since in our case, the analysis involves a fixed point iteration, the presentation as type systems does not appear to be a feasible solution.
The formalization of the CŻ definite assignment analysis we provide, sheds some light in particular on the complications generated by the goto and break statements (incompletely specified in [1]) and by the method calls with ref/out parameters - these are crucial differences with respect to Java. We also use the idea of data flow equations (see [11]) but due to the goto statement, the formalization cannot be done like in Java. For a method body without goto, however, the equations that characterize the sets of definitely assigned variables can be solved in a single pass. If goto statements are present, then the equations defined in our formalization do not specify in a unique way the sets of variables that have to be considered definitely assigned. For this reason, a fixed point computation is performed and the greatest sets of variables that satisfy the equations of the formalization are computed. Another difference with respect to Java is the presence of structs. Regarding the correctness of the analysis, we prove that, these sets of variables represent exactly the sets of variables assigned on all possible execution paths and in particular they are a safe approximation.

The rest of the paper is organized as follows. Section 2 introduces the data flow equations which formalize the CŻ definite assignment analysis while Section 3 shows that there always exists a maximal fixed point solution for the equations. In order to define the execution paths in a method body, the control flow graph is introduced in Section 4. The paper concludes in Section 5 with the proof of the correctness of the analysis, Theorem 1. A number of bugs in the Mono CŻ compiler [7] (version 0.26) which were discovered during the attempts to build the formalization of the definite assignment are included in the Appendix.

2 The data flow equations

In this section, we formalize the rules of definite assignment analysis from the CŻ Specification [11] §5.3 by data flow equations. Since the definite assignment analysis is an intraprocedural analysis, we restrict our formalization only to a given method meth. We use labels in order to identify the expressions and the statements. Labels are denoted by small Greek letters and are displayed as superscripts, for example, as in \( \alpha \exp \) or in \( \alpha \stm \). We will often refer to expressions and statements using their labels.

In order to precisely specify all the cases of definite assignment, static functions before, after, true, false and vars are computed at compile time. Note that true and false are only for boolean expressions. These functions assign sets of variables to each expression or statement \( \alpha \) and have the following meanings. before\((\alpha)\) contains the local variables definitely assigned before the evaluation of \( \alpha \) and after\((\alpha)\) the variables definitely assigned after the evaluation of \( \alpha \) when \( \alpha \) completes normally. true\((\alpha)\) and false\((\alpha)\) consist of the variables (in the scope of which \( \alpha \) is) definitely assigned after the evaluation of \( \alpha \) when \( \alpha \) evaluates to true and false, respectively. vars\((\alpha)\) contains the local variables in the scope of which \( \alpha \) is, i.e. the universal set with respect to \( \alpha \).

In order not to increase the number of equations, we skip those language constructs whose analysis is very similar to the one of the constructs dealt with explicitly in our framework; examples are alternative control structures (do, switch, foreach), pre- and postfix operators (++, --). On the other hand, we omit also the statements for and lock since they can be written in terms of constructs from our framework as observed by CŻ Specification in §5.3.3.9 and §8.12, respectively.

Before we state the data flow equations, we clarify some details concerning the definite assignment analysis for the struct type variables which represent a key difference with respect to Java. A struct type variable is considered definitely assigned if and only if each of its instance variables is considered definitely assigned [11] §5.3. We formalize this idea of propagating the definitely assigned status as follows. If a local variable \( \loc \) of a struct type is assigned, then all its instance fields \( \text{instfields}(\loc) \) get assigned and further, if there are struct type instance fields, then their instance fields get assigned as well and so on.

At the left example from Figure 1 the instance field \( y \) of \( p.x \) is definitely assigned before it is printed since \( p \) is definitely assigned.

The recursively defined set lookdown\((\loc)\) represents the local variables ‘derived’ from \( \loc \) which get assigned when \( \loc \) gets assigned.

\[
\text{lookdown}(\loc) = \begin{cases} 
\{ \loc \}, & \text{if } \loc \text{ is not of a struct type} \\
\{ \loc \} \cup \bigcup_{\text{field} \in \text{instfields}(\loc)} \text{lookdown}(\text{field}), & \text{if } \loc \text{ is of a struct type} 
\end{cases}
\]

At the same time, if a local variable \( \loc_1 \) which is an instance field of the struct type variable \( \loc_2 \)

1. gets assigned, and
struct P { public Q x; }
struct Q { public int y; }
class Test {
    public static void Main() {
        P p = new P();
        Console.WriteLine(p.x.y);
    }
}

struct P { public int x; public Q y; }
struct Q { public int u; public int v; }
class Test {
    public static void Main() {
        P p;
p.y.u = 2;
p.y.v = 3;
p.x = 1;
P r = p;
    }
}

Figure 1: The definite assignment analysis for struct type variables

2. each instance field of \( loc_2 \) other than \( loc_1 \) either is definitely assigned, i.e. is in the set \( \text{after}(\alpha) \) or is going to be assigned, i.e. is in the set \( \text{next} \): \( \{loc\} \) for a simple assignment to \( loc \) or \( \text{OutParams}(\arg_1, \ldots, \arg_k) \) for a method invocation with parameters \( \arg_1, \ldots, \arg_k \).

then the containing variable \( loc_2 \) gets assigned as well and the procedure is repeated for \( loc_2 \). In the right example from Figure 1 the struct type field \( p.y \) gets assigned when its instance field \( v \) gets assigned. Then the instance variable \( p \) gets assigned when its instance field \( p.x \) gets assigned. This is expressed by the computation of the set \( \text{lookup}(loc_1, \alpha, \text{next}) \).

\[
\text{lookup}(loc_1, \alpha, \text{next}) = \begin{cases} \{loc_1\} \cup \text{lookup}(loc_2, \alpha, \text{next}), & \text{if } loc_1 \text{ is an instance field of the } \\text{struct type variable } loc_2 \text{ and } \\text{instfields}(loc_2) \setminus \{loc_1\} \subseteq \text{after}(\alpha) \cup \text{next} \\ \{loc_1\}, & \text{otherwise} \end{cases}
\]

Therefore the set \( \text{struct}(loc, \alpha, \text{next}) \) contains the local variables that get definitely assigned when \( loc \) gets assigned where \( \alpha \) and the set \( \text{next} \) depends on the context - whether \( loc \) gets assigned through a simple assignment or through a method call.

\[
\text{struct}(loc, \alpha, \text{next}) = \text{lookup}(loc, \alpha, \text{next}) \cup \text{lookdown}(loc)
\]

We are now able to state all the data flow equations. A first equation is given by initial conditions: for the method body \( mb \) of \( \text{meth} \) we have \( \text{before}(mb) = \emptyset \). Actually we should consider the set of value and reference parameters of \( \text{meth} \) but there is no worry that an access to any of them could cause troubles since when \( \text{meth} \) is invoked they are supposed to be definitely assigned \( [1, 5.1] \).

For the other expressions and statements in \( mb \), instead of explaining how the functions are computed, we simply state the equations they have to satisfy. Table 1 contains the equations for boolean expressions. If \( \alpha \) is the constant \text{true}, then \( \text{false}(\alpha) = \text{vars}(\alpha) \) is a consequence of the definition of the \text{false} set and of the fact that \text{true} cannot evaluate to \text{false}. Similar arguments hold for \text{true}(\alpha) = \text{vars}(\alpha), if \( \alpha \) is the constant \text{false}. In addition, we have for all expressions in Table 1 the equation \( \text{after}(\alpha) = \text{true}(\alpha) \cap \text{false}(\alpha) \).

For a boolean expression \( \alpha \) which is not an instance of one of the expressions in Table 1 we have \( \text{true}(\alpha) = \text{after}(\alpha) \) and \( \text{false}(\alpha) = \text{after}(\alpha) \).

Table 2 lists the equations specific to arbitrary expressions where \( loc \) stands for a local variable and \( \text{lit} \) for a literal. The conditional expression is not the same like in Table 1, since it is not necessarily boolean. A \text{ref} parameter is used for ‘by reference’ parameter passing, in which the parameter acts as an alias for a caller-provided argument. An \text{out} parameter is similar to a \text{ref} parameter, except that the initial value of the caller-provided argument is not important. Concerning the definite assignment analysis, the \text{ref} arguments must be definitely assigned before the method invocation while the \text{out} arguments should not necessarily be assigned before the method is invoked. On the other hand, the \text{out} arguments must be definitely assigned when the method returns. Note that following a method invocation, not only the \text{out} parameters \( \text{OutParams}(\arg_1, \ldots, \arg_k) \) become definitely assigned, but eventually also struct type variables or instance fields of struct type variables.
\[ \text{the data flow equations} \]

<table>
<thead>
<tr>
<th>(e)</th>
<th>(\alpha)</th>
<th>(\beta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>(true(\alpha) = \text{before}(\alpha), false(\alpha) = \text{vars}(\alpha))</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>(false(\alpha) = \text{before}(\alpha), true(\alpha) = \text{vars}(\alpha))</td>
<td></td>
</tr>
<tr>
<td>(! \beta)</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), true(\alpha) = false(\beta), false(\alpha) = true(\beta))</td>
<td></td>
</tr>
<tr>
<td>((\beta_0 ? \gamma \gamma_1 : \delta \delta_2))</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), before(\gamma) = true(\beta), before(\delta) = false(\beta))</td>
<td>(true(\alpha) = true(\gamma) \cap true(\delta), false(\alpha) = false(\gamma) \cap false(\delta))</td>
</tr>
<tr>
<td>((\beta_1 &amp; &amp; \gamma \gamma_2))</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), before(\gamma) = true(\beta), true(\alpha) = true(\gamma),)</td>
<td>(false(\alpha) = false(\beta) \cap false(\gamma))</td>
</tr>
<tr>
<td>((\beta_1 \mid \gamma \gamma_2))</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), before(\gamma) = false(\beta), false(\alpha) = false(\gamma))</td>
<td>(true(\alpha) = true(\beta) \cap true(\gamma))</td>
</tr>
</tbody>
</table>

Table 1: Definite assignment for boolean expressions

<table>
<thead>
<tr>
<th>(e)</th>
<th>(\alpha)</th>
<th>(\beta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>(\text{after}(\alpha) = \text{before}(\alpha))</td>
<td></td>
</tr>
<tr>
<td>lit</td>
<td>(\text{after}(\alpha) = \text{before}(\alpha))</td>
<td></td>
</tr>
<tr>
<td>((\text{loc} = \beta\ e))</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), \text{after}(\alpha) = \text{after}(\beta) \cup \text{struct}(\text{loc}, \beta, {\text{loc}}))</td>
<td></td>
</tr>
<tr>
<td>((\text{loc} \text{ op} = \beta\ e))</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), \text{after}(\alpha) = \text{after}(\beta))</td>
<td></td>
</tr>
<tr>
<td>((\beta_0 ? \gamma \gamma_1 : \delta \delta_2))</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), before(\gamma) = true(\beta),)</td>
<td>(before(\delta) = false(\beta), after(\alpha) = after(\gamma) \cap after(\delta))</td>
</tr>
<tr>
<td>c.f</td>
<td>(\text{after}(\alpha) = \text{before}(\alpha))</td>
<td></td>
</tr>
<tr>
<td>ref (\beta\ e)</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), after(\alpha) = after(\beta))</td>
<td></td>
</tr>
<tr>
<td>out (\beta\ e)</td>
<td>(\text{before}(\beta) = \text{before}(\alpha), after(\alpha) = after(\beta))</td>
<td></td>
</tr>
<tr>
<td>((\text{c.m}\beta_1 \text{ arg}_1, \ldots, \beta_k \text{ arg}_k))</td>
<td>(\text{before}(\beta_1) = \text{before}(\alpha), before(\beta_{i+1}) = after(\beta_i), i = 1, k - 1,)</td>
<td>(after(\alpha) = after(\beta_k) \cup)</td>
</tr>
<tr>
<td></td>
<td>(\bigcup_{\text{arg}_i \in Out\text{Params}(\text{arg}_1, \ldots, \text{arg}_k)} \text{struct}(\text{arg}_i, \beta_k, \text{Out\text{Params}(\text{arg}_1, \ldots, \text{arg}_k)}))</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Definite assignment for arbitrary expressions

In cases not stated in Tables 1 and 2 if \(e\) is an expression with direct subexpressions \(\beta_1 \text{ e}_1, \ldots, \beta_n \text{ e}_n\), then the left-to-right evaluation scheme yields the general data flow equations:

\[ \text{before}(\beta_1) = \text{before}(\alpha), \text{before}(\beta_{i+1}) = \text{after}(\beta_i), i = 1, n - 1 \text{ and } \text{after}(\alpha) = \text{after}(\beta_n) \]

The equations specific for statements can be found in Table 3. We assume that the try statements are either try-catch or try-finally statements (see [2] for a justification of this assumption). Special attention is paid to the labeled statement. The set of variables definitely assigned before executing a labeled statement consists of the variables definitely assigned both after the previous statement and before each corresponding goto statement or after any of the finally blocks of try-finally statements in which the goto is embedded (if any). More exactly, this can be formalized as follows. For two statements \(\alpha\) and \(\beta\), we consider \(\text{Fin}(\alpha, \beta)\) to be the list \([\gamma_1, \ldots, \gamma_n]\) of finally blocks of all try-finally statements in the innermost to outermost order from \(\alpha\) to \(\beta\). Then we define the set \(\text{JoinFin}(\alpha, \beta)\) of
definitely assigned variables after the execution of all these finally blocks:

\[ \text{JoinFin}(\alpha, \beta) = \bigcup_{\gamma \in \text{Fin}(\alpha, \beta)} \text{after}(\gamma) \]

Further, we define the set goto for a statement \( \beta \). For a labeled statement \( \beta L : stm \), the set goto(\( \beta \)) is defined as follows:

\[ \text{goto}(\beta) = \bigcap_{\alpha \text{ goto L;}} (\text{before}(\alpha) \cup \text{JoinFin}(\alpha, \beta)) \]

where we take only the goto statements in the scope of \( \beta \). For all the other statements, as well for a labeled statement with no goto statements, goto(\( \beta \)) is the universal set vars(\( \beta \)). Now we are able to state the equation \( \text{before}(\beta_{i+1}) = \text{after}(\beta_i) \cap \text{goto}(\beta_{i+1}) \) from Table 3. In case of a labeled statement, the equation formalizes the above stated idea while for a non-labeled statement becomes \( \text{before}(\beta_{i+1}) = \text{after}(\beta_i) \).

The following example is a simplification of an example from the C\# Specification [1] §5.3.3.15:

```csharp
int i;
try { goto L;
try { goto L;
```
finally γ{i = 3;
βL:Console.WriteLine(i);
}

The explanations that come with the example in the C# Specification state that i is definitely assigned before β, i.e. i ∈ before(β). Our equation before(β) = after(δ) ∩ goto(β) led us to the same conclusion since goto(β) = before(α) ∪ after(γ) and i ∈ after(γ) ⊆ after(δ) (see the equations for a try-finally in Table 3). Surprisingly, the example is rejected by the C compilers of .NET Framework 1.0 and Rotor SSCLI [10]: we get the error that i is unassigned. In the meantime, this was fixed in .NET Framework 1.1 [11] but still exists in Rotor.

The following explanation holds for the equation after(α) = after(β) ∩ vars(α) corresponding to a block of statements: the local variables which are definitely assigned after the normal execution of the block are the variables which are definitely assigned after the execution of the last statement of the block. However, the variables must still be in the scope of a declaration. Thus, let us consider the example:

{int i; i = 1; } {int i; i = 2 * β i;

The variable i is not in after(α) since at the end of α, i is not in the scope of a declaration. Thus i ∉ before(β) and the block is rejected.

The idea for the equation which computes after(α) of a while statement α, is the same with that for a labeled statement. Similarly with the set goto, we define the set break(α) to be set of variables definitely assigned before all corresponding break statements (and possibly after appropriate finally blocks). This means that the set break(α) is defined by

\[ \text{break}(\alpha) = \bigcap_{\delta \text{break}} (\text{before}(\beta) \cup \text{JoinFin}(\beta, \alpha)) \]

where we take only the break statements for which α is the nearest enclosing while. If the while statement does not have any break statements, then we define break(α) = vars(α). With this definition of break(α), we have the equation for after(α) as stated in Table 3.

There is one more technical detail to be decided. Suppose we want to state the equation for after of a jump statement. Let α be the following statement:

if(b) γ{i = 1;} else δ return;

It is clear that, the variables definitely assigned after α are the variables definitely assigned after the then branch and since our equation takes the intersection of after(γ) and after(δ), it is obvious that one has to require the set-intersection identity for after(δ). That is why we adopt the convention that after(δ) is the universal set vars(α) for any jump statement α.

3 The maximal fixed point

The computation of the sets of definitely assigned variables from the data flow equations described in Section 2 is relatively straightforward. The key difference with respect to Java is the goto statement which brings more complexity to the analysis. Since the goto statement makes loops possible, the system of data flow equations does not have always a unique solution. Here is an example: if we consider a method which takes no parameters and has the following body

{int i = 1; βL: γgoto L;
}

then we have the following equations after(α) = {i}, before(β) = after(α) ∩ before(γ) and before(γ) = before(β). After some simplification we find that before(β) = {i} ∩ before(β) and therefore we get two solutions for before(β) (and also for before(γ)): ∅ and {i}. This is the reason we perform a fixed point iteration - which is not the case in Java. The set of variables definitely assigned after α is {i} and since β does not ‘unassign’ i, i is obviously assigned when we enter β. Consideration of the example and the definition of definitely assigned show that the most informative solution is {i} and therefore the solution we require is the maximal fixed point MFP. For computing this solution, there exist various algorithms (see 3).

Remark. Although the statements L: goto L; and while (true); look behaviorally similar, they are treated differently by the definite assignment analysis. Thus, if α denotes the labeled statement,
then the equation for $\text{before}(\alpha)$ implies recursion (as noticed above). If $\alpha$ is the while statement, then no equation corresponding to $\alpha$ involves recursion. The set $\text{after}(\alpha)$ of the above while statement can be computed according to the equations in a single step (i.e. with no fixed point iteration) as follows $\text{after}(\alpha) = \text{false}(\alpha) \cap \text{break}(\alpha) = \text{vars}(\alpha) \cap \text{vars}(\alpha) = \text{vars}(\alpha)$. The set $\text{before}(\alpha)$ is determined as for any regular statement using only the after set of the previous statement. Even if a while statement $\alpha$ has an associated continue statement $\gamma$, the equation for $\text{before}(\alpha)$ does not involve the continue but only the previous statement $\beta$. The reason is that, at the time the analysis is performed, the compiler is sure the continue is embedded in the while body and therefore the set $\text{before}(\gamma)$ includes the variables in $\text{after}(\beta)$ (if the continue is executed then necessarily $\beta$ should have been executed). This is not always true for a labeled statement since the associated goes are not necessarily embedded in the labeled block (see the last example of Section[2] and that is why they are involved in the equation for $\text{before}(\alpha)$.

In the rest of this section, we show that there always exists a maximal fixed point for our data flow equations. In order to prove the existence, one needs first to define the function $F$ which encapsulates the equations. For the domain and codomain of this function, we need the set $\text{Vars}(\text{meth})$ of all local variables from the method body $\text{mb}$. A simple inspection of the equations shows that they all have at the left side either a before, after, true or a false set and at the right side a combination of these kinds of sets and $\text{vars}$ sets. We define the function $F : D \rightarrow D$ with $D = \mathcal{P}(\text{Vars}(\text{meth}))^r$ such that $F(X_1, \ldots, X_r) = (Y_1, \ldots, Y_r)$, where $r$ is the number of equations and the sets $Y_i$ are defined by the data flow equations. For example in the case of an if-then-else statement, if the equation for the after set of this statement is the $i$-th data flow equation, then the set of variables $Y_i$ is defined by $Y_i = X_j \cap X_k$ where $j$ and $k$ are the indices of the equations for the after sets of the then and the else branch, respectively. Note that the sets $\text{vars}$ are interpreted as constants.

We define now the relation $\subseteq$ on $D$ to be the pointwise set inclusion relation: if $(X_1, \ldots, X_r) \in D$ and $(X'_1, \ldots, X'_r) \in D$, then we have $(X_1, \ldots, X_r) \subseteq (X'_1, \ldots, X'_r)$ if $X_i \subseteq X'_i$ for all $i = 1, \ldots, r$. We are now able to prove the following result:

**Lemma 1** $(D, \subseteq)$ is a finite lattice.

**Proof.** $D$ is finite since for a given method body we have a finite number of equations and local variables and on the other hand, $D$ is a lattice since it is a product of lattices: $(\mathcal{P}(\text{Vars}(\text{meth})), \subseteq)$ is a poset since the set inclusion is a partial order and for every two sets $X, Y \in \mathcal{P}(\text{Vars}(\text{meth}))$ there exists a lower bound $(X \cap Y)$ and an upper bound $(X \cup Y)$.

The following result will help us conclude the existence of the maximal fixed point.

**Lemma 2** The function $F$ is monotonic on $(D, \subseteq)$.

**Proof.** In order to prove the monotonicity of $F = (F_1, \ldots, F_r)$, it suffices to remark that the components $F_i$ are monotonic functions. This holds since they consist only of set intersections and unions which are monotonic (see the form of the equations).

The next result guarantees the existence of the maximal fixed point solution for our data flow equations:

**Lemma 3** The function $F$ has a unique maximal fixed point $\text{MFP} \in D$.

**Proof.** $(D, \subseteq)$ is a finite lattice (Lemma[1]) and therefore a complete lattice. But in a complete lattice, every monotonic function has a unique maximal fixed point (known also as the greatest fixed point). In our case, $F$ is monotonic (Lemma[2]) and the maximal fixed point $\text{MFP}$ is given by $\bigcap_k F^{(k)}(1_D)$. Here $1_D$ is the $r$-tuple $(\text{Vars}(\text{meth}), \ldots, \text{Vars}(\text{meth}))$, i.e. the top element of the lattice $D$.

From now on, for an expression or statement $\alpha$ we denote by $\text{MFP}_b(\alpha), \text{MFP}_a(\alpha), \text{MFP}_t(\alpha)$ and $\text{MFP}_f(\alpha)$ the components of $\text{MFP}$ corresponding to $\text{before}(\alpha), \text{after}(\alpha), \text{true}(\alpha)$ and $\text{false}(\alpha)$, respectively.
The main result we want to prove is that, for an arbitrary expression or statement, the sets of local variables MFP$_b$, MFP$_a$ (and MFP$_t$, MFP$_f$ for boolean expressions) correspond indeed to sets of definitely assigned variables, i.e. variables which are assigned on every possible execution path to the appropriate point. The considered paths are based on the control flow graph CFG. The nodes of the graph are actually points associated with every expression and statement. We suppose that every expression or statement $\alpha$ is characterized by an entry point $B(\alpha)$ and an end point $A(\alpha)$. Beside these two points, a boolean expression $\alpha$ has two more points: a true point $T(\alpha)$ (used when $\alpha$ evaluates to true) and a false point $F(\alpha)$ (used when $\alpha$ evaluates to false). The edges of the graph are given by the control transfer defined in the C$\#$ Specification [11, §8]. We show in Tables 4 and 5 the edges specific to each boolean and arbitrary expression, respectively. If the expression $\alpha$ is not an instance of one expression in these tables

<table>
<thead>
<tr>
<th>$\alpha$exp</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>$(B(\alpha), T(\alpha))$</td>
</tr>
<tr>
<td>false</td>
<td>$(B(\alpha), F(\alpha))$</td>
</tr>
<tr>
<td>$(! \beta e)$</td>
<td>$(B(\alpha), B(\beta)), (F(\beta), T(\alpha)), (T(\beta), F(\alpha))$</td>
</tr>
<tr>
<td>$(\beta e_0 \ ? \gamma e_1 : \delta e_2)$</td>
<td>$(B(\alpha), B(\beta)), (T(\beta), B(\gamma)), (F(\beta), B(\delta)), (T(\gamma), T(\alpha)), (F(\gamma), F(\alpha))$</td>
</tr>
<tr>
<td>$(\beta e_1 \ &amp;&amp; \gamma e_2)$</td>
<td>$(B(\alpha), B(\beta)), (T(\beta), B(\gamma)), (F(\beta), F(\alpha)), (T(\gamma), T(\alpha)), (F(\gamma), F(\alpha))$</td>
</tr>
<tr>
<td>$(\beta e_1 \</td>
<td>\</td>
</tr>
</tbody>
</table>

Table 4: Control flow for boolean expressions

<table>
<thead>
<tr>
<th>$\alpha$exp</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>$(B(\alpha), A(\alpha))$</td>
</tr>
<tr>
<td>lit</td>
<td>$(B(\alpha), A(\alpha))$</td>
</tr>
<tr>
<td>$(\text{loc} = \beta e)$</td>
<td>$(B(\alpha), B(\beta)), (A(\beta), A(\alpha))$</td>
</tr>
<tr>
<td>$(\text{loc op} = \beta e)$</td>
<td>$(B(\alpha), B(\beta)), (A(\beta), A(\alpha))$</td>
</tr>
<tr>
<td>$(\beta e_0 \ ? \gamma e_1 : \delta e_2)$</td>
<td>$(B(\alpha), B(\beta)), (T(\beta), B(\gamma)), (F(\beta), B(\delta)), (A(\gamma), A(\alpha)), (A(\delta), A(\alpha))$</td>
</tr>
<tr>
<td>c.f</td>
<td>$(B(\alpha), A(\alpha))$</td>
</tr>
<tr>
<td>ref$^\beta$exp</td>
<td>$(B(\alpha), B(\beta)), (A(\beta), A(\alpha))$</td>
</tr>
<tr>
<td>out$^\beta$exp</td>
<td>$(B(\alpha), B(\beta)), (A(\beta), A(\alpha))$</td>
</tr>
<tr>
<td>$c.m(\beta_1 \arg_1, \ldots, \beta_k \arg_k)$</td>
<td>$(B(\alpha), B(\beta_1)), (A(\beta_k), A(\alpha)), (A(\beta_i), B(\beta_{i+1})), i = 1, k-1$</td>
</tr>
</tbody>
</table>

Table 5: Control flow for arbitrary expressions

4 The control flow graph

The main result we want to prove is that, for an arbitrary expression or statement, the sets of local variables MFP$_b$, MFP$_a$ (and MFP$_t$, MFP$_f$ for boolean expressions) correspond indeed to sets of definitely assigned variables, i.e. variables which are assigned on every possible execution path to the appropriate point. The considered paths are based on the control flow graph CFG. The nodes of the graph are actually points associated with every expression and statement. We suppose that every expression or statement $\alpha$ is characterized by an entry point $B(\alpha)$ and an end point $A(\alpha)$. Beside these two points, a boolean expression $\alpha$ has two more points: a true point $T(\alpha)$ (used when $\alpha$ evaluates to true) and a false point $F(\alpha)$ (used when $\alpha$ evaluates to false). The edges of the graph are given by the control transfer defined in the C$\#$ Specification [11, §8]. We show in Tables 4 and 5 the edges specific to each boolean and arbitrary expression, respectively. If the expression $\alpha$ is not an instance of one expression in these tables
adds to the flow graph also the following edges:

\[(B(\alpha), B(\beta_1)), (A(\beta_1), B(\beta_{i+1})), i = 1, n - 1 \text{ and } (A(\beta_n), A(\alpha))\]

For every boolean expression \(\alpha\) in Table 4 we have the supplementary edges \((T(\alpha), A(\alpha))\) and \((F(\alpha), A(\alpha))\) which connect the boolean points of \(\alpha\) to the end point of \(\alpha\). These edges are necessary for the control transfer in cases when it does not matter whether \(\alpha\) evaluates to \text{true} or \text{false}. For example, if \(\beta\) is the method invocation \(c.m(\text{true})\) and \(\alpha\) is the argument \text{true}, then the control is transferred from the end point of the last argument - that is \(A(\alpha)\) - to the end point of the method invocation - that is \(A(\beta)\). But since in Table 4 we have no edge leading to \(A(\alpha)\), we need to define also the supplementary edge \((T(\alpha), A(\alpha))\).

For a boolean expression \(\alpha\) which is not an instance of any expression from Table 4 we add to the graph the edges \((A(\alpha), T(\alpha)), (A(\alpha), F(\alpha))\). They are needed if control is transferred from a boolean expression \(\alpha\) to different points depending on whether \(\alpha\) evaluates to \text{true} or \text{false}. For example, if \(\alpha\) is of the form \(\text{exp}_1 | \text{exp}_2\) and occurs in \(\beta((\text{exp}_1 | \text{exp}_2))\), then the control is transferred from \(F(\alpha)\) to \(T(\beta)\) (if \(\alpha\) evaluates to \text{false}) or from \(T(\alpha)\) to \(F(\beta)\) (if \(\alpha\) evaluates to \text{true}). The necessity of the edges \((A(\alpha), T(\alpha)), (A(\alpha), F(\alpha))\) arises since, so far we have defined for \(\text{exp}_1 | \text{exp}_2\) only edges to \(A(\alpha)\).

Table 5 introduces the edges of the control flow graph for each statement. Note that we assume that the boolean constant expressions are replaced by \text{true} or \text{false} in the abstract syntax tree. For example, we consider that \text{true} | \text{b} is replaced by \text{true} in the following if statement:

```
if \(\beta(\text{true} | \text{b}) \neq 1;
else \gamma(\text{int j = i;})
```

Although the new test (i.e. \text{true}) cannot evaluate to \text{false}, we still add to the graph the edge \((F(\beta), B(\gamma))\) since anyway the false point of \text{true} is not reachable (see Table 4).

In the presence of finally blocks, the jump statements \text{goto}, \text{break} and \text{continue} bring more complexity to the graph. Whenever such a jump statement exits one or more \text{try} blocks with associated \text{finally} blocks, the control is transferred first to the finally block (if any) of the innermost \text{try} statement. Further, if the control reaches the end point of the finally, then it is transferred to the next (with respect to the innermost to outermost order of the \text{try} statements) finally block and so on. If the control reaches the end point of the last finally block, then it is transferred to the target of the jump statement. For these control transfers we have special edges in our graph. But one needs to take care to some detail: these special edges cannot be used for paths other than those which connect the jump statement with its target. In other words, if a path uses such an edge, then necessarily the path contains the entry point of the jump statement. For this reason, we say that an edge \(e\) is conditionally by a point \(i\) with the meaning that \(e\) can be used only in paths that contain \(i\). If we do not make this restriction, then \([B(mb)B(\alpha_1)B(\alpha_2)B(\alpha_3)B(\alpha_4)B(\alpha_5)A(\alpha_5)B(\alpha_6)]\) would be a possible execution path to the labeled statement in the following method body

```
a1:try 
  a2 { 
    a3(a = (i = 1));
    goto L;
  } finally a5{}
```

\(a5:\text{Console.WriteLine}(i);\)

in the theoretical case when the evaluation of \(a_4\) would throw an exception. But this does not match the control transfer described in the C\# Specification.

The following sets introduce the above described edges. If \(\alpha\) and \(\beta\) are two statements and \(\text{Fin}(\alpha, \beta)\) is the list \(\gamma_1, \ldots, \gamma_n\), then the set \(\text{ThroughFin}_n(\alpha, \beta)\) consists of the edges \((B(\alpha), B(\gamma_1)), (A(\gamma_1), B(\beta)), (A(\gamma_1), B(\gamma_{i+1})), i = 1, n - 1 \text{ all conditioned by } B(\alpha) \text{ and the set } \text{ThroughFin}_n(\alpha, \beta)\) has the edges \((B(\alpha), B(\gamma_1)), (A(\gamma_1), A(\beta)), (A(\gamma_1), B(\gamma_{i+1})), i = 1, n - 1 \text{ all conditioned by } B(\alpha). \text{ If } \text{Fin}(\alpha, \beta)\) is empty, then the set \(\text{ThroughFin}_n(\alpha, \beta)\) has only the edge \((B(\alpha), B(\beta))\) while \(\text{ThroughFin}_n(\alpha, \beta)\) refers to the edge \((B(\alpha), A(\beta))\).

Note that in Table 6 for \text{goto} and \text{continue}, the set of edges \(\text{ThroughFin}_n\) is added to the graph, since after executing the finally blocks the control is transferred to the entry point of the labeled statement and while statement, respectively, while in case of \text{break} the set \(\text{ThroughFin}_n\) is considered, since at the end, the control is transferred to the end point of the while statement.
ThroughFin

<table>
<thead>
<tr>
<th>( \alpha \text{stm} )</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>( (\mathcal{B}(\alpha), \mathcal{A}(\alpha)) )</td>
</tr>
<tr>
<td>( (\beta \text{exp}) )</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha)) )</td>
</tr>
<tr>
<td>{ ( \beta_1 \text{stm}_1 \ldots \beta_n \text{stm}_n } )</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta_i)), (\mathcal{A}(\beta_i), \mathcal{A}(\alpha)), (\mathcal{A}(\beta), \mathcal{B}(\beta_{i+1})), i = 1, n - 1 )</td>
</tr>
<tr>
<td>if ( (\beta \text{exp}) \gamma \text{stm}_1 ) else ( \delta \text{stm}_2 )</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{B}(\delta)), (\mathcal{A}(\gamma), \mathcal{A}(\alpha)), (\mathcal{A}(\delta), \mathcal{A}(\alpha)) )</td>
</tr>
<tr>
<td>while ( (\beta \text{exp}) \gamma \text{stm} )</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{A}(\alpha)), (\mathcal{A}(\gamma), \mathcal{A}(\alpha)) )</td>
</tr>
<tr>
<td>L: ( \beta \text{stm} )</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha)) )</td>
</tr>
<tr>
<td>goto L;</td>
<td>( \text{ThroughFin}_b(\alpha, \beta), ) where ( \beta \text{L:stm} ) is the statement to which ( \alpha ) points</td>
</tr>
<tr>
<td>break;</td>
<td>( \text{ThroughFin}_b(\alpha, \beta), ) where ( \beta ) is the nearest enclosing while with respect to ( \alpha )</td>
</tr>
<tr>
<td>continue;</td>
<td>( \text{ThroughFin}_b(\alpha, \beta), ) where ( \beta ) is the nearest enclosing while with respect to ( \alpha )</td>
</tr>
<tr>
<td>return;</td>
<td>no edges</td>
</tr>
<tr>
<td>return ( \beta \text{exp} );</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)) )</td>
</tr>
<tr>
<td>throw;</td>
<td>no edges</td>
</tr>
<tr>
<td>throw ( \beta \text{exp} );</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)) )</td>
</tr>
<tr>
<td>try ( \beta \text{block} ) catch ( E_1 x_1 ) ( \gamma_1 \text{block}_1 )</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha)) )</td>
</tr>
<tr>
<td>:</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\gamma_i)), (\mathcal{A}(\gamma_i), \mathcal{A}(\alpha)) ) ( i = 1, n )</td>
</tr>
<tr>
<td>catch ( E_n x_n ) ( \gamma_n \text{block}_n )</td>
<td>( (\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{B}(\alpha), \mathcal{B}(\gamma)), (\mathcal{A}(\beta), \mathcal{B}(\gamma)) ) and ( (\mathcal{A}(\gamma), \mathcal{A}(\alpha)) ) conditioned by ( \mathcal{A}(\beta) )</td>
</tr>
</tbody>
</table>

Table 6: Control flow for statements

There are two more remarks concerning the try statement. Since in a try block can occur anytime a reason for abruption (e.g. an exception), we should have edges from every point in a try block to: every associate catch block, every catch of enclosing try statements (if the catch clause matches the type of the exception) and to every associate finally block (if none of the catch clauses matches the type of the exception). We do not consider all these edges, since from the point of view of the definite assignment analysis which is in particular an ‘over all paths’ analysis, it is equivalent to consider only one edge to the entry points of the catch and finally blocks - from the entry point of the try block (see Table 5).

The next remark is concerning the end point \( \mathcal{A}(\alpha) \) of a try-finally statement \( \alpha \). The C_{\text{spec}} Specification states in [8.10] that \( \mathcal{A}(\alpha) \) is reachable only if both end points of the try block \( \beta \) and finally block \( \gamma \) are reachable. The only edge to \( \mathcal{A}(\alpha) \) is \( (\mathcal{A}(\gamma), \mathcal{A}(\alpha)) \) and we know that the finally block can be reached either through a jump or through a normal completion of the try block. In case of a jump, if control
reaches the end point $A(\gamma)$ of the finally, then it is transferred further to the target of statement which generated the jump and not to $A(\alpha)$. This means that all paths to $A(\alpha)$ contain also the end point $A(\beta)$ of the try block. That is why we require that the edge $(A(\gamma), A(\alpha))$ is conditioned by $A(\beta)$ (see Table 6) - otherwise in the following example, $A(\alpha)$ would be reachable in our graph (under the assumption that $B(\alpha)$ is reachable):

```
  try \beta \{ goto L; \} finally \gamma \{
```

We define now the sets of valid paths to all points in the method body. We will not consider all the paths in the graph but only the valid paths - that is the paths $p$ for which the following is true: if $p$ uses a conditioned edge then it contains also the point which conditions the edge.

valid($[a_1, \ldots, a_n]$) $\equiv$ for every conditioned edge $(\alpha_i, \alpha_{i+1})$ $\exists j < i$ s.t. $\alpha_j$ conditions $(\alpha_i, \alpha_{i+1})$.

If $\alpha$ is an expression or a statement, then $\mathrm{path}_b(\alpha)$ and $\mathrm{path}_a(\alpha)$ are the sets of all valid paths from the entry point of the method body $\mathcal{B}(mb)$ to the entry point $\mathcal{B}(\alpha)$ and to the end point $A(\alpha)$ of $\alpha$, respectively.

\[
\mathrm{path}_b(\alpha) = \{(a_1, \ldots, a_n) \mid a_1 = \mathcal{B}(mb), a_n = \mathcal{B}(\alpha), (\alpha_i, \alpha_{i+1}) \in \text{CFG}, i = 1, n-1 \text{ and valid}([\alpha_1, \ldots, \alpha_n])\}
\]

\[
\mathrm{path}_a(\alpha) = \{(a_1, \ldots, a_n) \mid a_1 = \mathcal{B}(mb), a_n = A(\alpha), (\alpha_i, \alpha_{i+1}) \in \text{CFG}, i = 1, n-1 \text{ and valid}([\alpha_1, \ldots, \alpha_n])\}
\]

Similarly, if $\alpha$ is a boolean expression, then $\mathrm{path}_b(\alpha)$ and $\mathrm{path}_f(\alpha)$ are the sets of all valid paths from $\mathcal{B}(mb)$ to the true point $T(\alpha)$ and to the false point $F(\alpha)$ of $\alpha$, respectively.

In the proofs from the next section, we will use the following two notations. If $p$ is a path, then $p[i, j]$ is the subpath of $p$ which connects the point $i$ with the point $j$. Also, over the set of all paths, we define the operation $\odot$ to be the concatenation of the lists of points which define the paths.

5 The correctness of the analysis

We prove that, when a C$\#_c$ compiler relies on the sets $\mathrm{MFP}_b$, $\mathrm{MFP}_a$, $\mathrm{MFP}_t$ and $\mathrm{MFP}_f$ derived from the maximal fixed point of the data flow equations in Section 2, the risk of accessing the value of a local variable which is not initialized does not exist. In other words, the correctness of the analysis means that, if a local variable is in one of the four sets - that is the analysis infers the variable as definitely assigned at a certain program point, then this variable will actually be assigned at that point during every execution of the program. A variable is assigned on a path if the path contains an initialization of the variable or a catch clause whose exception variable is our variable. Due to the special treatment of struct type variables, we define in a recursive way what we mean by initialization. We say that a path $p$ contains an initialization of a local variable $loc$ if at least one of the following is true:

1. $p$ contains a simple assignment (not a compound assignment) to $loc$, or
2. $p$ contains a method invocation for which $loc$ is an out parameter, or
3. $loc$ is an instance field of a struct type variable $x$ and $p$ contains an initialization of $x$, or
4. $loc$ is of a struct type and $p$ contains initializations of each instance field of $loc$.

We prove actually more than the correctness. We show that the components of the maximal fixed point $\mathrm{MFP}$ are exactly (not only a safe approximation of) the sets of variables which are assigned on every possible path to the appropriate point. In order to formalize this, we define the following sets. If $\alpha$ is an arbitrary expression or statement, then $\mathrm{AP}_b(\alpha)$ and $\mathrm{AP}_a(\alpha)$ denote the sets of local variables in $\mathrm{vars}(\alpha)$ (the variables in the scope of which $\alpha$ is) which are assigned on every path in $\mathrm{path}_b(\alpha)$ and in $\mathrm{path}_a(\alpha)$, respectively.

\[
\mathrm{AP}_b(\alpha) = \{x \in \mathrm{vars}(\alpha) \mid x \text{ is assigned on every path } p \in \mathrm{path}_b(\alpha)\}
\]
\( \text{AP}_a(\alpha) = \{ x \in \text{vars}(\alpha) \mid x \text{ is assigned on every path } p \in \text{path}_a(\alpha) \} \)

For a boolean expression \( \alpha \), we have two more sets: \( \text{AP}_t(\alpha) \) and \( \text{AP}_f(\alpha) \) are defined similarly as above, but with respect to paths in \( \text{path}_t(\alpha) \) and \( \text{path}_f(\alpha) \), respectively.

The next result is used to prove Lemma 4.

**Lemma 4** For every expression or statement \( \alpha \), if \( \text{MFP}_b(\alpha) \subseteq \text{vars}(\alpha) \), then \( \text{MFP}_a(\alpha) \subseteq \text{vars}(\alpha) \). Moreover, if \( \alpha \) is a boolean expression, then we have also \( \text{MFP}_t(\alpha) \subseteq \text{vars}(\alpha) \) and \( \text{MFP}_f(\alpha) \subseteq \text{vars}(\alpha) \).

**Proof.** It is done for each expression and statement by induction over the abstract syntax tree, starting from the leaves. Thus, we will prove first the above stated implications for all possible leaves of the AST.

The statements considered leaves in an AST are the *empty-statement*, *goto L*, *break*, *continue*, *return*, and *throw*. For the last five, from the equations we obviously have \( \text{MFP}_a(\alpha) \subseteq \text{vars}(\alpha) \). For the *empty-statement*, this is true as well since our hypothesis is \( \text{MFP}_b(\alpha) \subseteq \text{vars}(\alpha) \).

In the next step, the so-called induction step, the implications for each expression and statement are proved, under the assumption that their children satisfy the implications. \( \square \)

The following lemma claims that, the MFP sets of an expression or statement \( \alpha \), consist of variables in the scope of which \( \alpha \) is.

**Lemma 5** For every expression or statement \( \alpha \) we have \( \text{MFP}_b(\alpha) \subseteq \text{vars}(\alpha) \) and \( \text{MFP}_a(\alpha) \subseteq \text{vars}(\alpha) \). Moreover, if \( \alpha \) is a boolean expression, then we have also \( \text{MFP}_t(\alpha) \subseteq \text{vars}(\alpha) \) and \( \text{MFP}_f(\alpha) \subseteq \text{vars}(\alpha) \).

**Proof.** We will show the above inclusions for all expressions and statements using an induction which starts from the root of the method body. So, under the assumption that a node of the abstract syntax tree AST satisfies the inclusions, we will prove that all its children satisfy the inclusions as well.

According to Lemma 4 it is enough to prove for all labels \( \alpha \): \( \text{MFP}_b(\alpha) \subseteq \text{vars}(\alpha) \).

For the method body, this is trivial: \( \text{MFP}_b(mb) \subseteq \text{vars}(mb) \) since \( \text{MFP}_b(mb) = \text{vars}(mb) = \emptyset \).

Lemma 4 is going to be used in the next step of the proof, which consists in showing for each expression or statement that, under the assumption \( \text{MFP}_b(\alpha) \subseteq \text{vars}(\alpha) \), each of its direct subexpressions or sub-statements \( \beta \) satisfies \( \text{MFP}_b(\beta) \subseteq \text{vars}(\beta) \). \( \square \)

The correctness of the definite assignment analysis in C# is proved in the following theorem, which claims that the analysis is a safe approximation.

**Theorem 1** *(safe approximation)* For every expression or statement \( \alpha \), the following relations are true: \( \text{MFP}_b(\alpha) \subseteq \text{AP}_b(\alpha) \) and \( \text{MFP}_a(\alpha) \subseteq \text{AP}_a(\alpha) \). Moreover, if \( \alpha \) is a boolean expression, then we have \( \text{MFP}_t(\alpha) \subseteq \text{AP}_t(\alpha) \) and \( \text{MFP}_f(\alpha) \subseteq \text{AP}_f(\alpha) \).

**Proof.** We consider the following definitions. The set \( \text{AP}_b^a(\alpha) \) is defined in the same way as \( \text{AP}_b(\alpha) \), except that we consider only the paths of length less than \( n \). Similarly, we define also the sets \( \text{AP}_t^a(\alpha) \), \( \text{AP}_f^a(\alpha) \), \( \text{AP}_t^b(\alpha) \) (analogously, we have definitions for the sets of paths \( \text{path}^a \)). According to these definitions, the following set equalities hold for an arbitrary \( \alpha \): \( \text{AP}_b(\alpha) = \bigcap_n \text{AP}_b^n(\alpha) \), \( \text{AP}_a(\alpha) = \bigcap_n \text{AP}_a^n(\alpha) \) and if \( \alpha \) is a boolean expression, then \( \text{AP}_t(\alpha) = \bigcap_n \text{AP}_t^n(\alpha) \) and \( \text{AP}_f(\alpha) = \bigcap_n \text{AP}_f^n(\alpha) \). Therefore to complete the proof, it suffices to show for every \( n \): if \( \alpha \) is an expression or statement, then \( \text{MFP}_b(\alpha) \subseteq \text{AP}_b^n(\alpha) \) and \( \text{MFP}_a(\alpha) \subseteq \text{AP}_a^n(\alpha) \) and in addition, if \( \alpha \) is a boolean expression, \( \text{MFP}_t(\alpha) \subseteq \text{AP}_t^n(\alpha) \) and \( \text{MFP}_f(\alpha) \subseteq \text{AP}_f^n(\alpha) \). This is done by induction on \( n \).

**Basis of induction:** \( \text{B}(\text{MB}) \) is the only path of length 1 (the entry point of the method body). Obviously, no local variable is assigned on this path and therefore we have \( \text{AP}_t^b(mb) = \emptyset \) which satisfies \( \text{MFP}_b(mb) \subseteq \text{AP}_b^b(mb) \) since from the equations \( \text{MFP}_b(mb) = \emptyset \). From the definition of \( \text{AP}_a^b \), we get \( \text{AP}_a^b(mb) = \text{vars}(mb) = \emptyset \) and from the equations of a block, we derive also \( \text{MFP}_a(mb) \subseteq \text{vars}(mb) \) and implicitly \( \text{MFP}_a(mb) \subseteq \text{AP}_a^b(mb) \). If \( \alpha \neq mb \), then \( \text{AP}_t^1(\alpha) = \text{AP}_a^b(\alpha) = \text{vars}(\alpha) \) and if \( \alpha \) is a boolean expression \( \text{AP}_t^1(\alpha) = \text{AP}_f^b(\alpha) = \text{vars}(\alpha) \). If we apply Lemma 5 then the basis of the induction is complete.
Induction step: The proof has the following pattern: we will show for all the expressions and statements from Tables 1, 2 and 3 the relations for MFPα and if it is case for MFPt and MFPf and for all their subexpressions and substatements, we will show the relations for MFPb. In this manner, all the relations are proved except that for MFPb(mb) which holds anyway since MFPb(mb) = ∅.

Whenever we consider an x in a MFP set of a certain expression or statement α, from the induction hypothesis x is also in the corresponding APn, and in particular in vars(α).

We prove first the relations corresponding to boolean expressions from Table 1.

**Case 1. a exp = true**

We want to prove for the boolean constant true the following subset relation: MFPt(α) ⊆ APn+1(α). Let x ∈ MFPt(α). Since MFP is a solution of the equations, we have MFPt(α) = MFPb(α) and consequently x ∈ MFPb(α). But according to the induction hypothesis, we have x ∈ APn(α). This means that x is assigned on every path in pathn(α). On the other hand, from the control flow graph CFG, we remark that the only way to reach the true point T(α) is through the edge (B(α), T(α)) which is equivalent with pathn+1(α) = pathn(α) ⊕ T(α). Because of this equality, we can conclude that x ∈ vars(α) is assigned on every path to T(α) of length at most n + 1, i.e. x ∈ APn+1(α).

On the other hand, we obviously have MFPf(α) ⊆ APn+1(α) since APn+1(α) = vars(α) - there is no path to F(α) - and from the equations we have MFPf(α) ⊆ vars(α).

**Case 2. a exp = false**

Similar to Case 1.

**Case 3. a exp = (1 β e)**

For the proof of MFPf(β) ⊆ APn(β), we consider an x ∈ MFPb(β). Because MFP is a fixed point of F, we have MFPb(β) = MFPb(α) and consequently x ∈ MFPb(α). From the induction hypothesis we get x ∈ APn(α). Since we have pathn+1(β) = pathn(α) ⊕ B(β), we can conclude that x ∈ APn+1(β). The proofs for the relations corresponding to MFPt(α) and MFPf(α) are similar.

**Case 4. a exp = (βe0 ? γe1 : βe2)**

The proofs for MFPb(β), MFPb(γ) and MFPb(δ) are similar with the proofs from Case 3. We are going to make now the proof corresponding to MFPt(α), namely MFPt(α) ⊆ APn+1(α). Let x be a local variable in MFPt(α). From the equations, we get x ∈ MFPt(γ) ∩ MFPt(δ) and further, from the induction hypothesis, we have x ∈ APn(γ) and x ∈ APn(δ). A simple inspection of CFG shows that there are only two ways to reach to T(α): either through the edge (T(γ), T(α)) or through the edge (T(δ), T(α)). This means that

pathn+1(α) = pathn(γ) ⊕ T(α) ∪ pathn(δ) ⊕ T(α)

which implies x ∈ APn+1(α). In the same way can be done also the proof for MFPf(α) ⊆ APn+1(α).

**Case 5. a exp = (∗e1 k & γ e2)**

Similar to Case 4.

**Case 6. a exp = (∗e1 l l γ e2)**

Similar to Case 4.

We consider now for proving the relations corresponding to an arbitrary expression from Table 2.

**Case 7. a exp = loc**

Similar to the proof corresponding to MFPb(β) in Case 3.

**Case 8. a exp = lit**

Similar to the proof corresponding to MFPb(β) in Case 3.

**Case 9. a exp = (loc = β e)**

The proof corresponding to MFPb(β) is similar with the proof for MFPb(β) in Case 3.

We want to prove now MFPα(α) ⊆ APn+1(α). If we consider an x ∈ MFPα(α), from the equations we get x ∈ MFPα(β) ∪ struct(loc, β, {loc}) where β is the expression which is assigned to α (see Table 2). If x ∈ MFPα(β), from the induction hypothesis we get x ∈ APn+1(β). The proof can be completed with the same arguments like before. We assume now that x is in the set struct(loc, β, {loc}) and then there two main cases to discuss:

13
We will consider now for proving the statements from Table 3.

• If \( x \in \text{lookdown}(\text{loc}) \), then either \( x \) is \( \text{loc} \) or \( x \in \bigcup_{\text{field} \in \text{instfields}(\text{loc})} \text{lookdown}(\text{field}) \) and \( \text{loc} \) is of a struct type. Further if \( x \) is \( \text{loc} \), then it is sure that \( x \) is assigned on any path to \( \mathcal{A}(\alpha) \), in particular also on paths of length at most \( n + 1 \), i.e. \( x \in \text{AP}^{n+1}_a(\alpha) \). In the other case, we get that there exists an instance field \( y \) of \( x \) such that \( x \in \text{lookdown}(y) \) and the procedure can be continued in this way. Since we obviously have to deal with a finite chain, \( x \) is one instance field in this chain. But since every path to \( \mathcal{A}(\alpha) \) contains an initialization of \( \text{loc} \), we say that it contains also an initialization of all the instance fields returned by the previous procedure (see the definition of \textit{initialization}). Therefore we get \( x \in \text{AP}^{n+1}_a(\alpha) \).

• If \( x \in \text{lookup}(\text{loc}, \beta, \{\text{loc}\}) \), then either \( x \) is \( \text{loc} \) and the proof is the same like before or \( \text{loc} \) is an instance field of a struct type \( y \) such that \( \text{instfields}(y) \subseteq \text{MFP}_a(\beta) \cup \{\text{loc}\} \) and \( x \in \text{lookup}(y, \beta, \{\text{loc}\}) \).

Let us suppose that the variable \( x \) is the struct variable \( y \). This implies that all the instance fields of \( x \) (except \( \text{loc} \)) are in the set \( \text{MFP}_a(\beta) \) and according to the induction hypothesis this means that the instance fields are assigned on every path to \( \mathcal{A}(\beta) \) of length at most \( n \). Since \( \text{loc} \) is assigned on every path to \( \mathcal{A}(\alpha) \) we obtain that \( x \) is in the set \( \text{AP}_a^{n+1}(\alpha) \).

If \( x \) is not the variable \( y \), then there exists a sequence \( y_1, \ldots, y_k \) of local variables such that \( x = y_k \), \( y = y_1 \), \( y_1, y_i \) is an instance field of the struct variable \( y_{i+1} \) for all \( i = 1, k - 1 \) and all the instance fields of \( y_i, i = 1, k \) except \( \text{loc} \) of \( y_1 \) and \( y_i, y_{i+1}, i = 1, k - 1 \) are in the set \( \text{MFP}_a(\beta) \) and implicitly in \( \text{AP}_a^{n+1}(\beta) \). With the same arguments as before, we can prove that \( y_1 \in \text{AP}_a^{n+1}(\alpha) \). In this manner, we can show by induction on \( i \) the desired result \( y_k \in \text{AP}_a^{n+1}(\alpha) \).

We have proved in both cases \( x \in \text{AP}_a^{n+1}(\alpha) \).

\textbf{Case 10.} \( \alpha \exp = (\text{loc op} = \beta e) \)

Similar to Case 9 with the difference that there is no case when \( x \) is in \( \text{struct}(\text{loc}, \beta, \{\text{loc}\}) \) since no initialization happens on the edge \( (\mathcal{A}(\beta), \mathcal{A}(\alpha)) \).

\textbf{Case 11.} \( \alpha \exp = (\beta e_0 ? \gamma e_1 : \delta e_2) \)

Similar to Case 4.

\textbf{Case 12.} \( \alpha \exp = c.f \)

Similar to the proof for \( \text{MFP}_b(\beta) \) in Case 3.

\textbf{Case 13.} \( \alpha \exp = \text{ref } \beta \exp \)

Similar to the proof for \( \text{MFP}_b(\beta) \) in Case 3.

\textbf{Case 14.} \( \alpha \exp = \text{out } \beta \exp \)

Similar to the proof for \( \text{MFP}_b(\beta) \) in Case 3.

\textbf{Case 15.} \( \alpha \exp = \text{c.m}(\beta_1 \arg_1, \ldots, \beta_k \arg_k) \)

The proofs corresponding to the arguments are similar with the proof for \( \text{MFP}_b(\beta) \) in Case 3.

We want show \( \text{MFP}_a(\alpha) \subseteq \text{AP}_a^{n+1}(\alpha) \). Let us consider a local variable \( x \) in the set \( \text{MFP}_a(\alpha) \). According to the equation satisfied by \( \text{MFP}_a(\alpha) \), we have also

\[ x \in \text{MFP}_a(\beta_k) \cup \bigcup_{\arg_i \in \text{OutParams}(\arg_1, \ldots, \arg_k)} \text{struct}(\arg_i, \beta_k, \text{OutParams}(\arg_1, \ldots, \arg_k)) \]

If \( x \in \text{MFP}_a(\beta_k) \), then the proof can be completed with the same arguments as in the corresponding proof for a simple assignment in Case 9. Let us assume now that there exists an \text{out} parameter \( \arg_i \) such that \( x \in \text{struct}(\arg_i, \beta_k, \text{OutParams}(\arg_1, \ldots, \arg_k)) \). The two possible cases \( x \in \text{lookdown}(\arg_i) \) and \( x \in \text{lookup}(\arg_i, \beta_k, \text{OutParams}(\arg_1, \ldots, \arg_k)) \) are treated similar with the corresponding cases of a simple assignment.

\textbf{Case 16.} The proofs corresponding to expressions which are not instances of expressions from Tables \[1\] and \[2\] make use of \textit{the general data flow equations} and follow the same pattern like the proof corresponding to \( \text{MFP}_b(\beta) \) in Case 3.

We will consider now for proving the statements from Table \[3\]
Case 17. $^astm =$;

Similar to the first proof in Case 3.

Case 18. $^astm = (^bexp;)$

Similar to the first proof in Case 3.

Case 19. $^astm = \{^bstm_1 \ldots ^bstm_n\}$

We want to prove now $MFP_b(\beta_{i+1}) \subseteq AP_{n+1}^b(\beta_{i+1})$ corresponding to an embedded statement. If we arbitrarily choose an $x$ in $MFP_b(\beta_{i+1})$, then we obtain $x \in MFP_a(\beta_i)$ and $x \in goto(\beta_{i+1})$ (from the flow equations). From the induction hypothesis, we get $x \in AP_a^n(\beta_i)$. Note that at this point, $goto(\beta_{i+1})$ depends only on MFP sets and on the control flow $CFG$.

- Let us consider first the case when $\beta_{i+1}$ is not a labeled statement. In this situation, we have $goto(\beta_{i+1}) = \text{vars}(\beta_{i+1})$.
  - If moreover $\beta_{i+1}$ is not a while statement, then there exists in $CFG$ only one edge to $B(\beta_{i+1})$, namely $(A(\beta_i), B(\beta_{i+1}))$. This means that
    \[
    \text{path}_{i+1}^a(\beta_{i+1}) = \text{path}_{i+1}^a(\beta_i) \cup B(\beta_{i+1})
    \]
    Since from the induction hypothesis $x \in AP_a^n(\beta_i)$, we get $x \in AP_{n+1}^b(\beta_{i+1})$.
  - If $\beta_{i+1}$ is a while statement, then there could be in $CFG$ more edges to $B(\beta_{i+1})$ (if there are continue statements pointing to this while).
    * If there are no continue statements corresponding to our while statement, then the proof is the same like before.
    * We want to show now that $x$ is assigned on every path to $B(\beta_{i+1})$ of length at most $n+1$ that pass through a continue statement and possibly through finally blocks of enclosing try-finally statements.
      If there exists such a path $p$ which contains a continue statement, then necessarily $p$ pass through $A(\beta_i)$ because the $CFG$ shows that it is not possible to jump into the while body where the continue is supposed to be embedded in. Since $x \in MFP_a(\beta_i)$, from the induction hypothesis we get $x \in AP_a^n(\beta_i)$. This means that $x$ is assigned on every path to $A(\beta_i)$ of length at most $n$. Now we are sure that $p$ assigns $x$.

Since we are sure that $x$ is assigned on each path to $B(\beta_{i+1})$ of length at most $n+1$, we can conclude $x \in AP_{n+1}^b(\beta_{i+1})$.

- Let us suppose now that $\beta_{i+1}$ is a labeled statement. Then like in the case of the while statement, there could be in $CFG$ more edges to $B(\beta_{i+1})$ (if there are goto statements pointing to our labeled statement).
  - If there are no corresponding goto statements, then the proof is the same like in the case of a while statement with no appropriate continue statements.
  - We want to show now that $x$ is assigned on every path $B(\beta_{i+1})$ of length at most $n+1$ that pass through a goto statement and possibly through finally blocks of enclosing try-finally statements.
    Let $p$ be such a path that pass through a goto statement $\gamma$. Since $x \in goto(\beta_{i+1})$, we get $x \in MFP_b(\gamma) \cup \text{JoinFin}(\gamma, \beta_{i+1})$.
    * If there are no finally blocks in $\text{Fin}(\gamma, \beta_{i+1})$, then $\text{JoinFin}(\gamma, \beta_{i+1}) = \emptyset$ and implicitly $x \in MFP_b(\gamma)$. Using the induction hypothesis, we obtain $x \in AP_b^n(\gamma)$. This means that $p$ which is of length at most $n+1$ and contains $B(\gamma)$ assigns $x$.
    * Let us suppose now that the list $\text{Fin}(\gamma, \beta_{i+1})$ is non-empty: $\text{Fin} = [\gamma_1, \ldots, \gamma_k]$. From the definition of $\text{JoinFin}(\gamma, \beta_{i+1})$, we get $x \in MFP_b(\gamma) \cup \bigcup_{j=1}^k MFP_a(\gamma_j)$
If \( x \in \text{MFP}_b(\gamma) \), then from the induction hypothesis we derive \( x \in \text{AP}_b^n(\gamma) \) and we are sure that \( p \) which pass through \( \mathcal{B}(\gamma) \) assigns \( x \).

If there is a finally block \( \gamma_j \) such that \( x \in \text{MFP}_a(\gamma_j) \), then the induction hypothesis implies \( x \in \text{AP}_a^n(\gamma_j) \). And since necessarily \( p \) pass through \( \mathcal{A}(\gamma_j) \), we are sure that \( p \) assigns \( x \).

Thus, we analyzed every possible path to \( \mathcal{B}(\beta_{i+1}) \) of length at most \( n+1 \) and we showed that each such a path assigns \( x \in \text{vars}(\beta_{i+1}) \), i.e \( x \in \text{AP}_b^{n+1}(\beta_{i+1}) \).

**Case 20.** \( \text{stm} = \text{if}(\beta \exp) \gamma \text{stm}_1 \) \text{else} \( \delta \text{stm}_2 \)

Similar to Case 4.

**Case 21.** \( \text{stm} = \text{while}(\beta \exp) \gamma \text{stm} \)

The proofs corresponding to \( \text{MFP}_a(\beta) \) and \( \text{MFP}_b(\gamma) \) are similar with the proof for \( \text{MFP}_b(\beta) \) in Case 3. The proof for \( \text{MFP}_a(\alpha) \) can be done in the same way as for the case of a labeled statement treated in Case 19.

**Case 22.** The proofs for the goto, break, continue, return, return exp, throw and throw exp statements (except those involving exp) are trivial: \( \text{MFP}_a(\alpha) \subseteq \text{AP}_a^{n+1}(\alpha) \) holds since from the induction hypothesis \( \text{MFP}_a(\alpha) \subseteq \text{AP}_a(\alpha) \) and on the other hand \( \text{AP}_a^n(\alpha) = \text{AP}_a^{n+1}(\alpha) = \emptyset \). The proofs for the points involving exp follow the same pattern like the proof for \( \text{MFP}_b(\beta) \) in Case 3.

**Case 23.** \( \text{stm} \) is a try-catch statement. The proofs for \( \text{MFP}_b(\alpha) \) and \( \text{MFP}_b(\gamma), i = 1, \ldots, n \) are similar with the proofs in Case 9. The proof for \( \text{MFP}_a(\alpha) \) follows the same pattern like the proof for \( \text{MFP}_a(\alpha) \) in Case 4.

**Case 24.** \( \text{stm} \) is a try-finally statement. The proof for \( \text{MFP}_b(\beta) \) is similar with the proof for \( \text{MFP}_b(\beta) \) in Case 3.

The next proof we are doing is of \( \text{MFP}_b(\gamma) \subseteq \text{AP}_b^{n+1}(\gamma) \). Let us consider an \( x \in \text{MFP}_b(\gamma) \). Following the data flow equations, we get \( x \in \text{MFP}_b(\gamma) \). The induction hypothesis implies \( x \in \text{AP}_b(\gamma) \). Now if we consider \( CFG \), there could be more edges leading to \( \mathcal{B}(\gamma) \): from the entry point of the corresponding try-finally statement \( \mathcal{B}(\alpha), \mathcal{B}(\gamma) \), from the end point of the corresponding try block \( \mathcal{A}(\beta), \mathcal{B}(\gamma) \), from a goto, break or continue statement \( \mathcal{B}(\delta), \mathcal{B}(\gamma) \) (within a conditioned path), from the end point of another finally block \( \mathcal{A}(\omega), \mathcal{B}(\gamma) \) (within a conditioned path).

We claim now that, independent of the last edge of an arbitrary path \( p \) to \( \mathcal{B}(\gamma) \), \( p \) pass through the beginning \( \mathcal{B}(\alpha) \) of the try-finally statement.

- If the last edge of \( p \) is \( (\mathcal{B}(\alpha), \mathcal{B}(\gamma)) \), then it is nothing to prove.
- If the last edge is \( (\mathcal{A}(\beta), \mathcal{B}(\gamma)) \), since the end point \( \mathcal{A}(\beta) \) can be reached only through \( \mathcal{B}(\alpha) \) (according to \( CFG \), it is not possible to jump into the try block), then the claim holds.
- For the case when the last edge of \( p \) is \( (\mathcal{B}(\delta), \mathcal{B}(\gamma)) \), the claim can be argued in the same way like above, because the respective jump statements are supposed to be embedded in the try block.
- If the last edge of \( p \) is a conditioned edge \( (\mathcal{A}(\omega), \mathcal{B}(\gamma)) \), then necessarily the finally block \( \omega \) (as well as the jump statement which triggered the conditioning) is embedded in our try block. It means that, in order to justify the claim, we can apply the same argument like above.

So all the paths to \( \mathcal{B}(\gamma) \) should pass through \( \mathcal{B}(\alpha) \) and since \( x \in \text{AP}_b^n(\alpha) \) we can be sure that \( x \in \text{vars}(\gamma) \) is assigned on every path to \( \mathcal{B}(\gamma) \) of length at most \( n+1 \), i.e. \( x \in \text{AP}_b^{n+1}(\gamma) \).

We want to prove now for a try-finally block: \( \text{MFP}_a(\alpha) \subseteq \text{AP}_a^{n+1}(\alpha) \). For an arbitrary \( x \in \text{MFP}_a(\alpha) \), we obtain from the data flow equations that \( x \in \text{MFP}_a(\beta) \) or \( x \in \text{MFP}_a(\gamma) \). Further from the induction hypothesis, we have \( x \in \text{AP}_b^n(\beta) \) or \( x \in \text{AP}_a^n(\gamma) \). As a consequence \( x \in \text{vars}(\beta) \) or \( x \in \text{vars}(\gamma) \), but because \( \text{vars}(\alpha) = \text{vars}(\beta) = \text{vars}(\gamma) \), we have for sure \( x \in \text{vars}(\alpha) \).

Note that if \( \mathcal{A}(\beta) \) is not reachable in \( CFG \), then every path to \( \mathcal{A}(\alpha) \) is not valid. Then we would have \( \text{AP}_a^{n+1}(\alpha) = \text{vars}(\alpha) \) and implicitly \( x \in \text{AP}_a^{n+1}(\alpha) \).

If \( \mathcal{A}(\beta) \) is reachable, then every valid path to \( \mathcal{A}(\alpha) \) should have as last edge \( (\mathcal{A}(\gamma), \mathcal{A}(\alpha)) \) and should implicitly pass through \( \mathcal{A}(\beta) \). That is why every valid path to \( \mathcal{A}(\alpha) \) pass through both \( \mathcal{A}(\beta) \) and \( \mathcal{A}(\gamma) \). Considering that \( x \in \text{AP}_a^n(\beta) \) or \( x \in \text{AP}_a^n(\gamma) \) and the above remark, we can conclude \( x \in \text{AP}_a^{n+1}(\alpha) \). □
We can prove actually more: the MFP solution is not only an approximation of AP but it is perfect (Theorem 2). For this, we need also the following theorem which states that the MFP solution contains the local variables which are initialized over all possible paths.

**Theorem 2** For every expression or statement \( \alpha \), the following relations are true: \( \text{AP}_b(\alpha) \subseteq \text{MFP}_b(\alpha) \) and \( \text{AP}_a(\alpha) \subseteq \text{MFP}_a(\alpha) \). Moreover, if \( \alpha \) is a boolean expression, then we have also \( \text{AP}_t(\alpha) \subseteq \text{MFP}_t(\alpha) \) and \( \text{AP}_f(\alpha) \subseteq \text{MFP}_f(\alpha) \).

**Proof.** Tarski’s fixed point theorem [13] states that MFP is the lowest upper bound (with respect to \( \subseteq \)) of the set \( \text{Ext}(F) = \{ X \in D \mid X \subseteq F(X) \} \). It suffices to show that the \( r \)-tuple consisting of the AP sets is an element of \( \text{Ext}(F) \) since MFP is in particular an upper bound of this set. Since \( \subseteq \) is the pointwise subset relation, the idea is to prove, for the data flow equations in Tables 1, 2 and 3 the left-to-right subset relations where instead of the sets before, after, true and false we have the sets \( \text{AP}_b \), \( \text{AP}_a \), \( \text{AP}_t \) and \( \text{AP}_f \), respectively.

We will make the proof first for the data flow equations of the boolean expressions from Table 1.

**Case 1.** \( ^* \text{exp} = \text{true} \)

We want to prove \( \text{AP}_t(\alpha) \subseteq \text{AP}_b(\alpha) \) and \( \text{AP}_f(\alpha) \subseteq \text{vars}(\alpha) \). If \( x \) is a local variable in \( \text{AP}_t(\alpha) \), then \( x \) is assigned on every path to \( T(\alpha) \) (see the definition of \( \text{AP}_t(\alpha) \)). Since in CFG all the paths to the true point \( T(\alpha) \) contains the entry point \( B(\alpha) \) and moreover there is no assignment to \( x \) in \text{true}, we deduce that \( x \) is assigned on every path to \( B(\alpha) \) and consequently \( x \in \text{AP}_b(\alpha) \), i.e. \( \text{AP}_t(\alpha) \subseteq \text{AP}_b(\alpha) \). The second subset relation holds because of the definition of \( \text{AP}_f(\alpha) \).

**Case 2.** \( ^* \text{exp} = \text{false} \)

Similar to Case 1.

**Case 3.** \( ^* \text{exp} = (! \beta) \)

We show that the following holds: \( \text{AP}_b(\beta) \subseteq \text{AP}_b(\alpha) \). Let \( x \) be an arbitrary local variable in \( \text{AP}_b(\beta) \), i.e. \( x \) is assigned on every path to \( B(\beta) \). We observe that all paths to \( B(\beta) \) contain the entry point \( B(\alpha) \). But since no local variable is assigned on the edge \( (B(\alpha), B(\beta)) \), we infer that \( x \) is assigned on every path to \( B(\alpha) \), i.e. \( x \in \text{AP}_b(\alpha) \).

Similar remarks hold also for the other two equations \( \text{AP}_t(\alpha) \subseteq \text{AP}_f(\beta) \) and \( \text{AP}_f(\alpha) \subseteq \text{AP}_t(\beta) \): all the paths to \( T(\alpha) \) and to \( F(\alpha) \) contain \( F(\beta) \) and \( T(\beta) \), respectively.

**Case 4.** \( ^* \text{exp} = (\beta e_0 \ ? \ ? e_1 : \ ? e_2) \)

The proofs for the first three equations are similar to Case 1. In order to prove \( \text{AP}_t(\alpha) \subseteq \text{AP}_t(\gamma) \cap \text{AP}_t(\delta) \), we notice that in CFG, the paths to \( T(\alpha) \) pass either through \( T(\gamma) \) or through \( T(\delta) \). If \( x \in \text{AP}_t(\alpha) \), then \( x \) is assigned on every path to \( T(\gamma) \) and on every path to \( T(\delta) \). Hence \( x \in \text{AP}_t(\gamma) \cap \text{AP}_t(\delta) \). The proof for \( \text{AP}_f(\alpha) \subseteq \text{AP}_f(\gamma) \cap \text{AP}_f(\delta) \) is similar.

**Case 5.** \( ^* \text{exp} = (\beta e_1 \ &\ & \ ? e_2) \)

Similar to Case 4.

**Case 6.** \( ^* \text{exp} = (\beta \ e_1 \ |\ | \ ? e_2) \)

Similar to Case 4.

We will prove now the relations corresponding to the arbitrary expressions in Table 2.

**Case 7.** \( ^* \text{exp} = \text{loc} \)

Similar to the first proof in Case 3.

**Case 8.** \( ^* \text{exp} = \text{lit} \)

Similar to the first proof in Case 3.

**Case 9.** \( ^* \text{exp} = (\text{loc} = \beta e) \)

The proof of \( \text{AP}_b(\beta) \subseteq \text{AP}_b(\alpha) \) is similar to the first proof in Case 3.

We want to prove \( \text{AP}_a(\alpha) \subseteq \text{AP}_a(\beta) \cup \text{struct}(\text{loc}, \beta, \{ \{ \text{loc} \} \} \cup \text{AP}_a(\beta) \). Let \( x \) be an arbitrary local variable in \( \text{AP}_a(\beta) \). Suppose that \( x \) is not in the set \( \text{AP}_a(\beta) \) otherwise the proof is complete. This means there exists a path \( p \) to \( A(\beta) \) on which \( x \) is not assigned. But \( x \) should be assigned on \( p \cup A(\alpha) \) since \( x \in \text{AP}_a(\alpha) \). Therefore, the assignment to \( \text{loc} \) to \( (A(\beta), A(\alpha)) \) determines the initialization of \( x \). According to the definition of initialization, this happens only in the following cases:
• the simple assignment to \( \text{loc} \) on the edge \((A(\beta), A(\alpha))\) is actually an assignment to \(x\), i.e. \(x = \text{loc}\). So \(x \in \text{struct}(\text{loc}, \beta, \{\text{loc}\})\).

• \(x_i\) is an instance field of a struct type variable \(x_{i+1}, i = 1, n - 1\) where \(x_1 = x\) and \(x_n = \text{loc}\). In this case, we get \(x \in \text{lookup}(\text{loc}) \subseteq \text{struct}(\text{loc}, \beta, \{\text{loc}\})\).

• \(x_i\) is an instance field of a struct type variable \(x_{i+1}, \) all the instance fields of \(x_{i+1}\) are assigned on \(p, i = 1, n - 1\) where \(x_1 = \text{loc}\) and \(x_n = x\). If all instance fields of \(x_{i+1}\) except \(x_i\) are in the set \(A(\beta\iota)\) for \(i = 1, n - 1\), then we get \(x \in \text{lookup}(\text{loc}, \beta, \{\text{loc}\}) \subseteq \text{struct}(\text{loc}, \beta, \{\text{loc}\})\). Let assume now there exists \(k\) such that an instance field \(y\) of \(x_k\) is not in \(A(\beta\iota); \) then there exists a path \(q\) (obviously other than \(p\)) on which \(y\) is not assigned. Clearly, \(y\) is not assigned also on \(q \oplus A(\alpha)\) and therefore also \(x\) is not assigned on \(q \oplus A(\alpha)\). But this contradicts \(x \in A(\alpha)\).

Case 10. \(\alpha \text{exp} = (\text{loc op} = \beta e)\)

Similar to the first proof in Case 3. This case is different from Case 9 since a compound assignment is not considered an initialization (because the variable is assumed to be already initialized). This is the reason there is no initialization on the edge \((B(\beta), B(\alpha))\).

Case 11. \(\alpha \text{exp} = (\beta_0 ? ^* e_1 : \delta e_2)\)

Similar to Case 4.

Case 12. \(\alpha \text{exp} = c.f\)

Similar to the first proof in Case 3.

Case 13. \(\alpha \text{exp} = \text{ref} \beta \text{exp}\)

Similar to the first proof in Case 3.

Case 14. \(\alpha \text{exp} = \text{out} \beta \text{exp}\)

Similar to the first proof in Case 3.

Case 15. \(\alpha \text{exp} = c.m(\beta_1 \text{arg}_1, \ldots, \beta_k \text{arg}_k)\)

The proofs of \(A(\beta_1) \subseteq A(\alpha)\) and \(A(\beta_{i+1}) \subseteq A(\beta_i), i = 1, n - 1\) are similar with the first proof in Case 3.

We want show:

\[
A(\alpha) = A(\beta_k) \cup \bigcup_{\text{arg}_i \in \text{OutParams}(\text{arg}_1, \ldots, \text{arg}_k)} \text{struct}(\text{arg}_i, \beta_k, \text{OutParams}(\text{arg}_1, \ldots, \text{arg}_k))
\]

Let us consider a variable \(x\) in \(A(\alpha)\) and let us suppose that \(x \notin A(\beta_k)\). Then there exists a path \(p\) to \(A(\beta_k)\) on which \(x\) is not assigned. On the other hand, \(x\) is assigned on \(p \oplus A(\alpha)\) since \(x \in A(\alpha)\). Therefore the initialization of \(x\) on the edge \((A(\beta_k), A(\alpha))\) is due to an initialization of an \text{out} parameter \(\text{arg}_i\). Further the proof is similar with the proof in the case of a simple assignment to \(\text{arg}_i\) (see Case 9).

Case 16. The proofs corresponding to the general data flow equations for the expressions which are not instances of expressions from Tables 1 and 2 are similar with first proof in Case 3.

We will consider now for proving the statements from Table 3.

Case 17. \(\alpha \text{stm} = ;\)

Similar to the first proof in Case 3.

Case 18. \(\alpha \text{stm} = (\beta \text{exp};)\)

Similar to the first proof in Case 3.

Case 19. \(\alpha \text{stm} = \{\beta_1 \text{stm}_1, \ldots, \beta_n \text{stm}_n\}\)

A special case of a block is our method body. In order to prove \(A(\beta) \subseteq \emptyset\), it is enough to notice that \(A(\beta) = \emptyset\). This is indeed so because the only path to \(B(\beta)\) is \([B(\beta)]\) on which there is no initialization of any local variable.

For an arbitrary block of statements, the proof of \(A(\beta) \subseteq A(\alpha)\) is similar to the first proof in Case 3. To show \(A(\alpha) \subseteq A(\beta_n) \cap \text{vars}(\alpha)\), it suffices to notice that we have \(A(\alpha) \subseteq A(\beta_n)\) (proved with similar arguments like in Case 3) and \(A(\alpha) \subseteq \text{vars}(\alpha)\) (from the definition of \(A(\alpha)\)).
We want to prove now the subset relation for an arbitrary index $i$: $\text{AP}_b(\beta_{i+1}) \subseteq \text{AP}_a(\beta_i) \cap \text{goto}(\beta_{i+1})$

Note that here, $\text{goto}(\beta_{i+1})$ depends only on the AP sets and on the control flow graph $CFG$. We have two possible cases:

- If the statement $\beta_{i+1}$ is not a labeled statement, then according to the definition of $\text{goto}(\beta_{i+1})$, we have $\text{goto}(\beta_{i+1}) = \text{vars}(\beta_{i+1}) = \text{vars}(\beta_i)$ and since $\text{AP}_a(\beta_i) \subseteq \text{vars}(\beta_i)$, we need to prove only $\text{AP}_b(\beta_{i+1}) \subseteq \text{AP}_a(\beta_i)$ which is obvious since, all the paths to $A(\beta_i)$ are, except the edge $(A(\beta_i), B(\beta_{i+1}))$, also paths to $\beta_{i+1}$ and no variable is assigned on this edge. Note that it does not matter for the above proof whether $A(\beta_i)$ is reachable or not in $CFG$.

- If the statement $\beta_{i+1}$ is a labeled statement $L: \text{stmt}$, then we could have more edges in $CFG$ leading to $B(\beta_{i+1})$. Let us consider a variable $x \in \text{AP}_b(\beta_{i+1})$, i.e. there exists an assignment to $x$ on every path leading to $B(\beta_{i+1})$. Exactly like in other case we can prove $\text{AP}_b(\beta_{i+1}) \subseteq \text{AP}_a(\beta_i)$. In order to show $x \in \text{goto}(\beta_{i+1})$, we need to prove that $x$ is in the set $\text{AP}_b(\gamma) \cup \text{JoinFin}(\gamma, \beta_{i+1})$ for every associated $\text{goto}$ $L$ statement.

Depending on whether there are $\text{goto}$ statements pointing to $\beta_{i+1}$ we have the following case distinction:

- If there is no associated $\text{goto}$ statement, then obviously we have $x \in \text{goto}(\beta_{i+1})$ since in this case $\text{goto}(\beta_{i+1}) = \text{vars}(\beta_{i+1})$.

- If there exists at least one associated $\text{goto}$ statement $\gamma$.
  - If $\text{B}(\gamma)$ is not reachable in $CFG$, then $\text{path}_b(\gamma) = \emptyset$ and consequently we get $x \in \text{AP}_b(\gamma) \cup \text{JoinFin}(\gamma, \beta_{i+1})$ because $\text{AP}_b(\gamma) = \text{vars}(\gamma) \supseteq \text{vars}(\beta_{i+1})$. The last subset relation holds because, in $CFG$ a $\text{goto}$ statement should be always in the scope of the corresponding labeled statement.
  - If $\text{B}(\gamma)$ is reachable in $CFG$, then let $p$ be an arbitrary path to $B(\gamma)$.

We will make now another case distinction.

- If there are no $\text{finally}$ blocks from $\gamma$ to $\beta_{i+1}$, i.e. $\text{Fin}(\gamma, \beta_{i+1}) = \emptyset$, then we get $\text{JoinFin}(\gamma, \beta_{i+1}) = \emptyset$ (since the union of an empty collection of sets is defined to be the empty set). This means that, in this case, it suffices to show $x \in \text{AP}_b(\gamma)$. In this case, we have in $CFG$ also the edge $(B(\gamma), B(\beta_{i+1}))$ that is defined by the set $\text{ThroughFin}_b(\gamma, \beta_{i+1})$. So, $p \oplus B(\beta_{i+1})$ is a path to $B(\beta_{i+1})$ and since $x \in \text{AP}_b(\beta_{i+1})$, we are sure that there is an assignment to $x$ also on $p$. Because $p$ was arbitrarily chosen, we get $x \in \text{AP}_b(\gamma)$.

- Let us consider now the case when there are $\text{finally}$ blocks from $\gamma$ to $\beta_{i+1}$, i.e. $\text{Fin}(\gamma, \beta_{i+1}) = \{\gamma_1, \ldots, \gamma_k\}$. Accordingly, also the edges

  \[(B(\gamma), B(\gamma_1)), (A(\gamma_j), B(\beta_{i+1})), (A(\gamma_j), B(\gamma_{j+1})), j = 1..k-1\]

from the set $\text{ThroughFin}_b(\gamma, \beta_{i+1})$ are added to $CFG$.

We will prove $x \in \text{AP}_b(\gamma) \cup \text{JoinFin}(\gamma, \beta_{i+1})$ by reduction to absurdum. Let us assume that $x \notin \text{AP}_b(\gamma) \cup \text{JoinFin}(\gamma, \beta_{i+1})$. This is equivalent with $x \notin \text{AP}_a(\gamma)$ and $x \notin \text{AP}_b(\gamma_j)$ for all $j = 1..k$. This means that there exist the paths $p_j \in \text{path}_b(\gamma_j)$, $p_j \in \text{path}_b(\gamma_j)$ for $j = 1..k$ such that $x$ is not assigned on any of these paths. A simple inspection of $CFG$ shows that, necessarily the point $B(\gamma_j)$ occurs in the path $p_j$ for every $j = 1..k$ since it is not possible to 'jump' into a $\text{finally}$ block. We want to prove now that the following

  \[p_0 \oplus p_1[B(\gamma_1), A(\gamma_1)] \oplus \ldots \oplus p_k[B(\gamma_k), A(\gamma_k)] \oplus B(\beta_{i+1})\]

is a valid path to $B(\beta_{i+1})$. The only problem that could arise is concerning the conditioned edges. Remember that the edges conditioned by a certain $\text{goto}$, $\text{break}$ or $\text{continue}$ statement can be used only in paths or subpaths that contain the entry point of the respective jump statement.

The use of the edges $(B(\gamma), B(\gamma_1)), \ldots, (A(\gamma_k), B(\beta_{i+1}))$ is correct as long as our path contains $B(\gamma)$. 

19
Let us consider one of the subpaths $p_j[B(\gamma_j), A(\gamma_j)]$ contained in the above defined path. If it contains a conditioned edge, then since the conditioned edges connect jump statements with finally blocks, we are sure that these embedded finally blocks are contained in our finally block $\gamma_j$. Then the respective jump statement is embedded into the try blocks (corresponding to the conditioned connected finally blocks) which necessarily should be embedded in our finally block $\gamma_i$ (this is an immediate consequence of the C grammar). So the jump statement is embedded into our finally block. Considering that the subpath $p_j[B(\gamma_j), A(\gamma_j)]$ contains conditioned edges, we get that also $p_j$ uses the same conditioned edges and since we assumed that $p_j$ is a valid path, necessarily $p_j$ should contain the entry point of the respective jump statement which, as we proved above, is embedded in our finally block, and consequently appears in the subpath $p_j[B(\gamma_j), A(\gamma_j)]$. This means that this subpath is valid. Obviously this is true for all the above considered subpaths. The above defined path to $B(\beta_{i+1})$ does not assign $x$. Obviously this contradicts $x \in AP_b(\beta_{i+1})$ and therefore our assumption is wrong. Hence, we obtain the desired $x \in AP_b(\gamma) \cup \text{JoinFin}(\gamma, \beta_{i+1})$.

There is one more special case, the while statements. In the same way like in case of a labeled statement, also for $\beta_{i+1}$while statement we may have several edges to $B(\beta_{i+1})$ if there are associated continue statements embedded into $\beta_{i+1}$. But we do not care for the proof of this theorem about possible edges coming from continue statements: we need only to prove $AP_b(\beta_{i+1}) \subseteq AP_b(\beta_i)$ which can be done exactly like in the case of a regular statement, considering only the paths via $A(\beta_i)$.

This completes the proofs of the data flow equations for a block of statements.

Case 20. $\alpha \text{stm} = \text{if } (\beta \text{exp}) \gamma \text{stm}_1 \text{ else } \delta \text{stm}_2$

Similar to Case 4.

Case 21. $\alpha \text{stm} = \text{while}(\beta \text{exp}) \gamma \text{stm}$

The proofs corresponding to the first two equations are similar with first proof in Case 3. We want now to prove the third equation:

$$AP_a(\alpha) \subseteq AP_f(\beta) \cap break(\alpha)$$

where the set $break(\alpha)$ depends only on the AP sets and on the control flow $CFG$. This proof is similar with corresponding proof for a labeled statement in Case 19 with the following observations: the role of $A(\beta_i)$ is played now by $F(\beta_i), B(\beta_{i+1})$ by $A(\alpha)$ and the goto statements are replaced by break statements.

Case 22. The proofs for the goto, break, continue, return, return exp, throw and throw exp statements (except those involving exp) are trivial: $AP_a(\alpha) \subseteq \text{vars}(\alpha)$ is true because of the definition of $AP_a(\alpha)$. The proofs for the points involving exp follow the same pattern like the first proof in Case 3.

Case 23. $\alpha \text{stm}$ is a try-catch statement. The following subset relations

$$AP_b(\beta) \subseteq AP_b(\alpha), AP_b(\gamma_i) \subseteq AP_b(\alpha) \cup \{x_i\}, i = 1, n$$

$$AP_a(\alpha) \subseteq AP_a(\beta), AP_a(\alpha) \subseteq AP_a(\gamma_i), i = 1, n$$

are satisfied since in $CFG$ we have defined the edges

$$(B(\alpha), B(\beta)), (A(\beta), A(\alpha)), (B(\alpha), B(\gamma_i)), (A(\gamma_i), A(\alpha)), i = 1, n$$

This completes the proofs for the try-catch statements.

Case 24. $\alpha \text{stm}$ is a try-finally statement. The proofs for the first two relations are similar to Case 23.

We want to prove now $AP_a(\alpha) \subseteq AP_a(\beta) \cup AP_a(\gamma)$. Note that there is no regular edge $(A(\beta), A(\alpha))$ or $(A(\gamma), A(\alpha))$ in $CFG$. There is only the conditioned edge $(A(\gamma), A(\alpha))$.

We will make the proof by reduction to absurdum. Let us suppose there exists an $x \in AP_a(\alpha)$ such that $x \notin AP_a(\beta)$ and $x \notin AP_a(\gamma)$. Then there exist the paths $p_1 \in path_a(\beta)$ and $p_2 \in path_a(\gamma)$ that do not
contain any assignments to \( x \). The construction of \( CFG \) shows that the point \( B(\gamma) \) occurs in any path to \( A(\gamma) \) (it is not possible to jump into the \textit{finally} block), in particular in \( p_2 \).

The following
\[
p_1 \oplus p_2[B(\gamma), A(\gamma)] \oplus A(\alpha)
\]
is a valid path to \( A(\alpha) \) since the use of the \textit{conditioned} edge is OK as long as the path contains \( A(\beta) \) and the subpath \( p_2[B(\gamma), A(\gamma)] \) is valid (with the same proof like in the case of a labeled statement).

Under our assumption, the above constructed path does not contain any assignments to \( x \) and this contradicts \( x \in AP_a(\alpha) \). It means that our assumption was wrong and we can conclude that \( AP_a(\alpha) \subseteq AP_a(\beta) \cup AP_a(\gamma) \).

The following result is then an obvious consequence of Theorem 1 and Theorem 2.

**Theorem 3** The maximal fixed point solution of the data flow equations in Tables 1, 2, 3 represents the sets of local variables which are assigned over all possible execution paths. More exactly, for every expression or statement \( \alpha \), the following are true: \( AP_b(\alpha) = MFP_b(\alpha) \) and \( AP_a(\alpha) = MFP_a(\alpha) \). Moreover, if \( \alpha \) is a boolean expression: \( AP_t(\alpha) = MFP_t(\alpha) \) and \( AP_f(\alpha) = MFP_f(\alpha) \).

### 6 Conclusion and future work

In this paper, we have formalized the definite assignment analysis of \( C_\# \) by data flow equations. Since the equations do not always have a unique solution, we defined the outcome of the analysis as the solution of a fixed point iteration. We proved that there exists always a maximal fixed point solution MFP. We showed the correctness of the analysis, i.e. MFP is a \textit{safe approximation} of the sets of variables assigned over all possible paths. This is a key property for the type safety of \( C_\# \) whose formalization is part of future work as well as proving the correctness of \( C_\# \) compilers. This paper is part of a research project focusing on formalizing and verifying important aspects of \( C_\# \). So far, we have an ASM model for the operational semantics of \( C_\# \) in [2]. During the attempts to build this model, there were discovered in [4] a few discrepancies between the \( C_\# \) Specification and different implementations of \( C_\# \).

**Acknowledgment** We gratefully acknowledge partial support of the related work [2] by a Microsoft grant within the ROTOR project during the year 2002-2003.

### References


http://msdn.microsoft.com/net/sscli/ and http://www.sscli.net/


matics 5, 1955.

7 Appendix

During the attempts to build the formalization of the definite assignment analysis in C♯ a few bugs were discovered in the Mono C♯ compiler [7] (version 0.26).

- **Test1**: Mono C♯ compiler claims that $i$ is unassigned.

```
using System;

class Test1 {
    public static void Main() {
        bool b = false;
        int i;
        if (b) {
            i = 1;
            goto L;
        }
        return;
        L: Console.WriteLine(i);
    }
}
```

At the entry of the labeled statement, $i$ should be considered definitely assigned since it is assigned on the single path leading to the labeled statement - that is the path which contains the `goto` statement.

- **Test2**: Mono C♯ compiler accepts the program.

```
using System;

class Test2 {
    public static void Main() {
        bool b = false;
        int i;
        if (b && (i = 1) >= 0)
            Console.WriteLine(i);
        else
            Console.WriteLine(i);
    }
}
```

If $b$ evaluates to `false`, then the test $(b && (i = 1) >= 0)$ is evaluated to `false` and the second operand of `&&`, i.e. $(i = 1) >= 0$) is not anymore evaluated. So in this case $i$ is not assigned. This means that $i$ is not definitely assigned at the entry point of the `else` branch. Although $i$ is not definitely assigned at that point, the Mono C♯ compiler does not reject the program.

- **Test3**: Mono C♯ compiler accepts the program.

```
using System;

class Test3 {
    public static void Main() {
        bool b = true;
        int i;
        if (b || (i = 1) >= 0)
            Console.WriteLine(i);
        else
```

23
If \( b \) evaluates to `true`, then the test \((b \lor (i = 1) \geq 0)\) is evaluated to `true` and the second operand of `\|`, i.e. \((i = 1) \geq 0\) is not anymore evaluated. So in this case \( i \) is not assigned. This means that \( i \) is not definitely assigned at the entry if the `then` branch. Although \( i \) is not definitely assigned at that point, the Mono C\# compiler does not reject the program.

- **Test4:** Mono C\# compiler claims that \( i \) is unassigned.

```csharp
using System;

class Test4 {
    public static void Main() {
        bool b = false;
        int i;
        while (true) {
            if (b) {
                i = 1;
                break;
            }
        }
        Console.WriteLine(i);
    }
}
```

It is not clear at all from the C\# Specification, what are the definitely assigned variables after the execution of the `while` statement in case there exists an appropriate `break` statement. Our equations answer precisely to this question: the variables which are definitely assigned both before the `break` statement and after the evaluation to `false` of the constant `true` (this happens for all local variables), in particular \( i \). So the example should not be rejected by the Mono C\# compiler.

- **Test5:** Mono C\# compiler claims that \( i \) is unassigned.

```csharp
using System;

class Test5 {
    public static void Main() {
        bool b = false;
        int i;
        while (b ? true : true);
        Console.WriteLine(i);
    }
}
```

According to C\# Specification [\[5.3.3.7\]], the variables which are definitely assigned after the execution of the `while` statement are the variables which are assigned after the evaluation to `false` of the test expression `b?true:true` - that variables which are definitely assigned after the evaluation to `false` of the constant `true`. This means that all the local variables in the scope of which `while` is should be considered as definitely assigned, in particular \( i \). That is why, the Mono C\# compiler should not reject the program. Note that the end point of `while` is considered to be reachable and therefore it is not the same situation like in the next example.

- **Test6:** Mono C\# compiler claims that \( i \) is unassigned.

```csharp
using System;
```
```csharp
class Test6 {
    public static void Main() {
        int i;
        if (true)
            return;
        else
            Console.WriteLine(i);
    }
}
```

According to C# Specification [1 §5.3.3.1], `i` is definitely assigned at the beginning of any unreachable statement. In our case, the `else` branch is unreachable since the test is the constant `true` [1 §8.7.1]. This means that `i` is definitely assigned at the beginning of the `else` branch and the program should not be rejected by the Mono C# compiler.