JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 9
Special issue: .NET Technologies 2004 workshop, Pilsen

The Correctness of the Definite Assignment Analysis in C♯

Nicu G. Fruja, Computer Science Department, ETH Zürich, Switzerland

The compilation of C♯ requires a flow analysis to ensure that every local variable is *definitely assigned* when any access to its value occurs. A variable is definitely assigned at a use of its value if every execution path leading to that use contains an assignment to the variable. Since local variables are uninitialized by default, this prevents access to uninitialized memory which is a crucial ingredient for the type safety of C♯. We formalize the rules of the definite assignment analysis of C♯ 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 a certain program point, then the variable will actually be initialized at that point during every execution of the program. We actually prove more than correctness: we show that the solution of the analysis is a perfect solution (and not only a *safe approximation*).

1 INTRODUCTION

In C♯ local variables are not initialized by default, unlike static and instance fields. Therefore, in order to ensure type-safety — an expression's value at runtime is always a subtype of its static type — a C♯ compiler must guarantee that all local variables are assigned to before their value is used. The C♯ compiler enforces this *definite assignment rule* by a static flow analysis. Since the problem is undecidable in general, the C♯ Language Specification [1, §5.3] contains a definition of a decidable subclass of the set of variables that get assigned at run-time. The static analysis guarantees that there is an initialization to a local variable on every possible execution path before the variable is read.

In this paper, we provide a formalization of the definite assignment analysis in C♯ that helps us to prove the analysis correct. So far, the definite assignment analysis of the Java compiler has been formalized with data flow equations in the work of Stärk et al. [6] and related to the problem of generating verifiable bytecode from legal Java source code programs.

The formalization of the C♯ definite assignment analysis emphasizes in particular the complications caused by the `goto` and `break` statements (incompletely specified in [1]) and by method calls with `ref`/`out` parameters — these are crucial differences with respect to Java (notice that Java has a `break L;` statement which has no corresponding statement in C♯). The struct type variables represent another difference with respect to Java. We consider the treatment of the struct type variables

Cite this article as follows: Nicu G. Fruja: "The Correctness of the Definite Assignment Analysis in C♯", in *Journal of Object Technology*, vol. 3, no. 9, October 2004, Special issue: .NET Technologies 2004 workshop, pp. 29–52, http://www.jot.fm/issues/issue_2004_10/article2

in detail. We use the idea of data flow equations but due to the `goto` statement, the formalization cannot be done as for Java. For a method body without a `goto` statement, 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 uniquely determine 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. 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*. A number of bugs in the Rotor SSCLI [3] and Mono [4] (version 0.26) C♯ compilers were discovered during the attempts to build the formalization of the definite assignment. We present three of them; the rest are described in the Appendix of [5].

The rest of the paper is organized as follows. Section 2 introduces the data flow equations which formalize the C♯ 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.

2 THE DATA FLOW EQUATIONS

In this section, we formalize the rules of definite assignment analysis from the C♯ Specification [1, §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 prefixed superscripts, for example, as in ${}^{\alpha}exp$ or in ${}^{\alpha}stm$. We often refer to expressions and statements using their labels only.

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 defined for boolean expressions. These functions assign sets of variables to each expression or statement α and have the following meanings:

- $before(\alpha)$ contains the variables definitely assigned before the evaluation of α;
- $after(\alpha)$ contains the variables definitely assigned after the evaluation of α when α completes normally;
- $true(\alpha)$ consists of the variables (in the scope of which α is located) definitely assigned after the evaluation of α when α evaluates to `true`;
- $false(\alpha)$ consists of the variables (in the scope of which α is located) definitely assigned after the evaluation of α when α evaluates to `false`;

The sets *true* and *false* are needed because of the conditional operators `&&` and `||`, as we show in an example. The set $vars(\alpha)$ contains the local variables in the scope of which α is located, i.e. the universal set with respect to α.

For clarity of presentation, we skip those language constructs whose analysis is very similar to the constructs dealt with explicitly in our framework; examples are alternative control structures (`do`, `switch`, `foreach`) and the pre- and postfix operators (`++`, `--`). We omit also the statements `for` and `lock` since they can be written in terms of constructs from our framework as observed by the C♯ Specification in [§5.3.3.9] and [§8.12], respectively.

Struct type variables. From the point of view of the definite assignment analysis, the struct type variables in C♯ represent a key difference with respect to Java. We will treat them separately to simplify the proofs. The interested reader can find in [5] a formalization with equations that include the struct types. We point out in Section 3 how they affect the sets of definitely assigned variables the C♯ compiler relies on in order to analyze programs. Also, we show in Section 5 that allowing variables of struct types does not affect the correctness of the analysis. In the rest of the paper we state explicitly whenever we include struct type variables.

We are now able to state all the data flow equations. A first equation is given by the method's initial conditions: for the method body *mb* of *meth* we have $before(mb) = \emptyset$. Conceptually, the set $before(mb)$ contains the value and reference parameters of *meth* since they are assumed to be definitely assigned when *meth* is invoked [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 (including for completeness the literals `true` and `false`). If α is the constant `true`, then $false(\alpha) = vars(\alpha)$ as a consequence of the definition of the *false* set and of the fact that `true` cannot evaluate to `false`. Similar arguments hold for $true(\alpha) = vars(\alpha)$ when α is the constant `false`. We need the sets *true* and *false* since the evaluation of boolean expressions involving the conditional operators `&&` and `||` does not necessarily require the evaluation of all their subexpressions. Consider the following expression:

$^{\alpha}$`(b && (i = 1) >= 0) ? true :` $^{\gamma}$ `i > 0`

If `b` evaluates to `false`, then the test `(b && (i = 1) >= 0)` immediately evaluates to `false` and its second operand, i.e. `(i = 1) >= 0` is never evaluated. So in this case `i` is not assigned; on the other hand, a necessary condition for the test to be evaluated to `true` is that both operands of α are evaluated. Therefore, the C♯ compiler is sure that `i` is assigned only if α evaluates to `true`. Formally, this means `i` $\notin false(\alpha)$ and `i` $\in true(\alpha)$. Consequently `i` is not considered definitely assigned before evaluating γ and the compiler should reject this example. Unfortunately the Mono C♯ compiler [4] incorrectly accepts it, as it does also for the other conditional operator, `||`.

${}^{\alpha}exp$	the data flow equations
`true`	$true(\alpha) = before(\alpha), false(\alpha) = vars(\alpha)$
`false`	$false(\alpha) = before(\alpha), true(\alpha) = vars(\alpha)$
(`!` ${}^{\beta}e$)	$before(\beta) = before(\alpha), true(\alpha) = false(\beta),$ $false(\alpha) = true(\beta)$
(${}^{\beta}e_0$ `?` ${}^{\gamma}e_1$ `:` ${}^{\delta}e_2$)	$before(\beta) = before(\alpha), before(\gamma) = true(\beta),$ $before(\delta) = false(\beta), true(\alpha) = true(\gamma) \cap true(\delta),$ $false(\alpha) = false(\gamma) \cap false(\delta)$
(${}^{\beta}e_1$ `&&` ${}^{\gamma}e_2$)	$before(\beta) = before(\alpha), before(\gamma) = true(\beta),$ $true(\alpha) = true(\gamma), false(\alpha) = false(\beta) \cap false(\gamma)$
(${}^{\beta}e_1$ `||` ${}^{\gamma}e_2$)	$before(\beta) = before(\alpha), before(\gamma) = false(\beta),$ $false(\alpha) = false(\gamma), true(\alpha) = true(\beta) \cap true(\gamma)$

Table 1: Definite assignment for boolean expressions

In addition, we have for all expressions in Table 1 the equation $after(\alpha) = true(\alpha) \cap false(\alpha)$. For any boolean expression α which is not an instance of one of the expressions in Table 1, we have $true(\alpha) = after(\alpha)$ and $false(\alpha) = after(\alpha)$.

Table 2 lists the equations specific to arbitrary expressions where *loc* stands for a local variable and *lit* for a literal. Note that the table contains another equation for the conditional expression when its value is not a boolean. The equation for the explicitly boolean expression collects additional information. If the boolean conditional is treated as an arbitrary expression, then the equation for $after(\alpha)$ would still be correct — it can be derived from the other equations for the boolean expressions.

In C♯ a `ref` parameter is used for "by reference" parameter passing in which the parameter acts as an alias for a caller-provided argument. An `out` parameter is similar to a `ref` parameter except that the initial value of the caller-provided argument is not important. The `ref` arguments must be definitely assigned before the method invocation, while the `out` arguments are not necessarily assigned before the method is invoked. However, the `out` arguments must be definitely assigned when the method returns. Note the equation for $after(\alpha)$ of a method invocation in Table 2: the `out` arguments, denoted as $OutParams(arg_1, \ldots, arg_k)$, get definitely assigned. We take into consideration static methods as well as instance methods and we do not care if we have recursive method calls since the definite assignment analysis is intraprocedural.

For expressions that do not appear in Tables 1 and 2 (e.g. e_1`|`e_2, e_1`+`e_2), if ${}^{\alpha}exp$ is an expression with *direct subexpressions* ${}^{\beta_1}e_1, \ldots, {}^{\beta_n}e_n$, then the left-to-right

${}^{\alpha}exp$	the data flow equations
loc	$after(\alpha) = before(\alpha)$
lit	$after(\alpha) = before(\alpha)$
$(loc = {}^{\beta}e)$	$before(\beta) = before(\alpha)$, $after(\alpha) = after(\beta) \cup \{loc\}$
$(loc\ op = {}^{\beta}e)$	$before(\beta) = before(\alpha), after(\alpha) = after(\beta)$
$({}^{\beta}e_0\ ?\ {}^{\gamma}e_1\ :\ {}^{\delta}e_2)$	$before(\beta) = before(\alpha), before(\gamma) = true(\beta)$, $before(\delta) = false(\beta), after(\alpha) = after(\gamma) \cap after(\delta)$
$c.f$	$after(\alpha) = before(\alpha)$
ref ${}^{\beta}exp$	$before(\beta) = before(\alpha),\ after(\alpha) = after(\beta)$
out ${}^{\beta}exp$	$before(\beta) = before(\alpha),\ after(\alpha) = after(\beta)$
$c.m({}^{\beta_1}arg_1, \ldots, {}^{\beta_k}arg_k)$	$before(\beta_1) = before(\alpha)$, $before(\beta_{i+1}) = after(\beta_i),\ i = \overline{1, k-1}$, $after(\alpha) = after(\beta_k) \cup OutParams(arg_1, \ldots, arg_k)$

Table 2: Definite assignment for arbitrary expressions

evaluation scheme yields the following *general data flow equations*:

$$before(\beta_1) = before(\alpha), before(\beta_{i+1}) = after(\beta_i),\ i = \overline{1, n-1}, after(\alpha) = after(\beta_n)$$

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 [11] for a justification of this assumption).

Notice that for a block of statements α we have the equation $after(\alpha) = after(\beta_n) \cap vars(\alpha)$: 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. Consider the example:

```
{ α { int i; i = 1;} {int i; i = 2 * β i;}}
```

The variable `i` is not in $after(\alpha)$ since at the end of α `i` is not in the scope of a declaration. Thus `i` $\notin before(\beta)$ and the block is rejected.

For the equation $before(\beta_{i+1}) = after(\beta_i) \cap goto(\beta_{i+1})$, special attention is given to the case when β_{i+1} is a labeled statement. A key point is that if a `goto` embedded in a `try` block (of a `try-finally` statement) points to a labeled statement which is not embedded in the `try` block, then the `finally` block has to be executed (before the labeled statement). Thus the set of variables definitely assigned before

${}^{\alpha}stm$	the data flow equations
`;`	$after(\alpha) = before(\alpha)$
$({}^{\beta}exp\,\texttt{;})$	$before(\beta) = before(\alpha), after(\alpha) = after(\beta)$
$\{{}^{\beta_1}stm_1 \ldots\ {}^{\beta_n}stm_n\}$	$before(\beta_1) = before(\alpha)$, $after(\alpha) = after(\beta_n) \cap vars(\alpha)$, $before(\beta_{i+1}) = after(\beta_i) \cap goto(\beta_{i+1})$, $i = \overline{1, n-1}$
`if` $({}^{\beta}exp)\ {}^{\gamma}stm_1$ `else` ${}^{\delta}stm_2$	$before(\beta) = before(\alpha), before(\gamma) = true(\beta)$, $before(\delta) = false(\beta)$, $after(\alpha) = after(\gamma) \cap after(\delta)$
`while` $({}^{\beta}exp)\ {}^{\gamma}stm$	$before(\beta) = before(\alpha), before(\gamma) = true(\beta)$, $after(\alpha) = false(\beta) \cap break(\alpha)$
`goto L;`	$after(\alpha) = vars(\alpha)$
`break;`	$after(\alpha) = vars(\alpha)$
`continue;`	$after(\alpha) = vars(\alpha)$
`return;`	$after(\alpha) = vars(\alpha)$
`return` ${}^{\beta}exp$`;`	$before(\beta) = before(\alpha), after(\alpha) = vars(\alpha)$
`throw;`	$after(\alpha) = vars(\alpha)$
`throw` ${}^{\beta}exp$`;`	$before(\beta) = before(\alpha), after(\alpha) = vars(\alpha)$
`try` ${}^{\beta}block$ `catch(`$E_1\ x_1$`)` ${}^{\gamma_1}block_1$ ⋮ `catch(`$E_n\ x_n$`)` ${}^{\gamma_n}block_n$	$before(\beta) = before(\alpha)$, $before(\gamma_i) = before(\alpha) \cup \{x_i\}, i = \overline{1, n}$, $after(\alpha) = after(\beta) \cap \bigcap_{i=1}^{n} after(\gamma_i)$
`try` ${}^{\beta}block_1$ `finally` ${}^{\gamma}block_2$	$before(\beta) = before(\alpha), before(\gamma) = before(\alpha)$, $after(\alpha) = after(\beta) \cup after(\gamma)$

Table 3: Definite assignment for statements

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). We formalize this as follows. For two statements α and β we consider $Fin(\alpha, \beta)$ to be the list $[\gamma_1, \ldots, \gamma_n]$ of `finally` blocks of all `try-finally` statements in the innermost to outermost order from α to β. Then we define the set $JoinFin(\alpha, \beta)$ of

definitely assigned variables after the execution of all these `finally` blocks:

$$JoinFin(\alpha, \beta) = \bigcup_{\gamma \in Fin(\alpha,\beta)} after(\gamma)$$

Further, we provide the definition for the set *goto* of a statement β. If β is a labeled statement $^{\beta}$`L:` *stm*, the set $goto(\beta)$ is defined as follows:

$$goto(\beta) = \bigcap_{^{\alpha}\texttt{goto L;}} (before(\alpha) \cup JoinFin(\alpha, \beta))$$

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

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

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

The C♯ Specification explains that in this example `i` is definitely assigned before β, i.e. `i` $\in before(\beta)$. Our equation $before(\beta) = after(\delta) \cap goto(\beta)$ leads us to the same conclusion. To compute the set $goto(\beta)$, we need the list $Fin(\alpha, \beta) = [\gamma]$ and the set $JoinFin(\alpha, \beta) = after(\gamma)$. We have:

$$goto(\beta) = before(\alpha) \cup\ JoinFin(\alpha, \beta) = before(\alpha) \cup\ after(\gamma)$$

and `i` $\in after(\gamma) \subseteq after(\delta)$ (see the equations for a `try-finally` in Table 3). This means that `i` $\in after(\delta) \cap goto(\beta) = before(\beta)$. Surprisingly, the example is rejected by the C♯ compilers of .NET Framework 1.0 and Rotor SSCLI [3]: we get the error that `i` is unassigned. In the meantime this problem is fixed in .NET Framework 1.1 [2] but still exists in Rotor.

Although in the next method body `i` should be considered definitely assigned before β the example is rejected by the Mono C♯ compiler [4].

```
 int i; bool b = false;
 if (b) { i = 1; αgoto L; }
δreturn;
βL:Console.WriteLine(i);
```

Note that `i` $\in before(\beta)$ since the set $before(\beta)$ is computed as follows:
$before(\beta) = after(\delta) \cap goto(\beta) = vars(\delta) \cap before(\alpha) = \{\texttt{i}, \texttt{b}\} \cap \{\texttt{i}\} = \{\texttt{i}\}$.

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

$$break(\alpha) = \bigcap_{\beta\texttt{break;}} (before(\beta) \cup JoinFin(\beta, \alpha))$$

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 $break(\alpha) = vars(\alpha)$. With this definition of $break(\alpha)$ we have the equation for $after(\alpha)$ as stated in Table 3.

Abnormal termination. Finally we consider the equations for abnormally terminating statements. 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 δreturn;
```

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 $after(\gamma)$ and $after(\delta)$ it is obvious that one has to require the set-intersection identity for $after(\delta)$. That is why we adopt the convention that $after(\alpha)$ is the universal set $vars(\alpha)$ for any jump statement α. Consider also the next method body:

```
δint i = 1;
βL:;
 try {
   int j = 2;
   try { αgoto L; }
   finally γ1{ int k = 3; ωthrow new Exception(); }
 } finally γ2{}
```

In our formalization we get: $before(\beta) = after(\delta) \cap goto(\beta) = after(\delta) \cap (before(\alpha) \cup JoinFin(\alpha, \beta)) = after(\delta) \cap (before(\alpha) \cup after(\gamma_1) \cup after(\gamma_2))$. Note that in the computation of $JoinFin(\alpha, \beta)$ we do not care whether γ_1 and γ_2 complete normally. In our case γ_1 does not complete normally, but we still perform our computations with γ_1 and γ_2. The set $after(\gamma_1)$ is $vars(\omega) \cap vars(\gamma_1) = \{\texttt{i}, \texttt{j}, \texttt{k}\} \cap \{\texttt{i}, \texttt{j}\} = \{\texttt{i}, \texttt{j}\}$. Note also that $after(\gamma_1)$ involved in the equation of $before(\beta)$ contains also `j`, while β is not in the scope of a declaration of `j`, i.e. $\texttt{j} \notin vars(\beta)$. There is no worry since in the equation of $before(\beta)$ all the sets that might contain variables declared in "deeper" scopes (like `j`) are intersected with $after(\delta)$ which is supposed to contain only variables from $vars(\delta) = vars(\beta) = \{\texttt{i}\}$ (β and δ are at the same nesting level). These details become more clear in Lemmas 4 and 5 in Section 5. Whenever a statement does not complete normally the set of variables considered definitely assigned after its evaluation will be a universal set (see also the proofs of Theorems 1 and 2 from Section 5).

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 allows to encode loops, the system of data flow equations does not have always a unique solution. Here is an example: consider a method which takes no parameters and has the following body:

$\{^{\alpha}$`int i = 1;` $^{\beta}$`L:` $^{\gamma}$`goto L;`$\}$

We have the following equations $after(\alpha) = \{$`i`$\}$, $before(\beta) = after(\alpha) \cap before(\gamma)$ and $before(\gamma) = before(\beta)$. After some simplification, we find that $before(\beta) = \{$`i`$\} \cap before(\beta)$. Therefore we get two solutions for $before(\beta)$ (and also for $before(\gamma)$): $\emptyset$ and $\{$`i`$\}$. This is the reason we perform a fixed point iteration — which is not necessary in the definite assignment analysis for Java. The set of variables definitely assigned after α is $\{$`i`$\}$; since β does not 'unassign' `i`, `i` is obviously assigned when we enter β. The example and the definition of *definitely assigned* indicate that the most informative solution is $\{$`i`$\}$ and therefore the solution we require is the maximal fixed point MFP. For computing this solution various algorithms exist (see e.g. [8]).

Remark. Although the statements `L: goto L;` and `while (true);` are behaviorally similar, they are treated differently by the definite assignment analysis. If α denotes the labeled statement, then the equation for $before(\alpha)$ implies recursion (as noticed above). If α is the `while` statement above, then no equation corresponding to α involves recursion. The set $after(\alpha)$ 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 $after(\alpha) = false(\alpha) \cap break(\alpha) = vars(\alpha) \cap vars(\alpha) = vars(\alpha)$. The set $before(\alpha)$ is determined as for any regular statement using only the *after* set of the previous statement. Even if a `while` statement α has an associated `continue` statement γ, the equation for $before(\alpha)$ does not involve the `continue` but only the previous statement β. The reason is that, at the time the analysis is performed, the compiler is sure that the `continue` is embedded in the `while` body and therefore the set $before(\gamma)$ includes the variables in $after(\beta)$ (if the `continue` is executed then necessarily β should have been executed). This is not always the case for a labeled statement since the associated `goto`s are not necessarily embedded in the labeled block (see the last example of Section 2). That is why they are involved in the equation for $before(\alpha)$.

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 its existence we first define the function F which encapsulates the equations. For the domain and codomain of this function we need the set $Vars(meth)$ of all local variables from the method body 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 *vars* sets. We define the function $F : D \to D$ with $D = \mathcal{P}(Vars(meth))^r$

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 where the *vars* sets are interpreted as constants. 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.

We define now the relation $\sqsubseteq$ on D to be the pointwise set inclusion relation:

Definition 1 *If* $(X_1, \ldots, X_r)$ *and* $(X'_1, \ldots, X'_r)$ *are elements in* D*, then we have* $(X_1, \ldots, X_r) \sqsubseteq (X'_1, \ldots, X'_r)$ *if* $X_i \subseteq X'_i$ *for all* $i = \overline{1, r}$.

We are now able to prove the following result:

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

Proof. D is finite since for a given method body we have a finite number of equations and local variables. On the other hand, D is a lattice since it is a product of lattices: $(\mathcal{P}(\mathit{Vars}(\mathit{meth})), \subseteq)$ is a *poset* since the set inclusion is a partial order and for every two sets $X, Y \in \mathcal{P}(\mathit{Vars}(\mathit{meth}))$ 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* $(D, \sqsubseteq)$.

Proof. 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* $MFP \in D$.

Proof. $(D, \sqsubseteq)$ 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 *MFP* is given by $\bigcap_k F^{(k)}(1_D)$. Here 1_D is the r-tuple $(\mathit{Vars}(\mathit{meth}), \ldots, \mathit{Vars}(\mathit{meth}))$, i.e. the top element of the lattice D. □

¿From now on for an expression or statement α we denote by $\mathrm{MFP}_b(\alpha)$, $\mathrm{MFP}_a(\alpha)$, $\mathrm{MFP}_t(\alpha)$ and $\mathrm{MFP}_f(\alpha)$ the components of the maximal fixed point *MFP* corresponding to $\mathit{before}(\alpha)$, $\mathit{after}(\alpha)$, $\mathit{true}(\alpha)$ and $\mathit{false}(\alpha)$, respectively.

Struct type variables. Up to now we have not considered struct type variables. However, our analysis can easily be extended to struct type variables. First we observe that a struct type variable is considered definitely assigned if and only if all its instance fields are definitely assigned [1, §5.3] (this is because a struct value can be seen as a tuple consisting of its instance fields).
If a local variable *loc* of a struct type gets assigned, then all of its instance fields are considered definitely assigned and if there are struct type instance fields, then their instance fields get assigned as well and so on. In the following example we assume that `A.m` is a static method that takes an `out` parameter of type `P`. The instance field `y` of `p.x` is definitely assigned before it is printed since `p` gets definitely assigned after the call of `A.m`.

```
struct P {
  public Q x;
}

struct Q {
  public int y;
}
```

```
class Test {
  static void Main() {
    P p;
    A.m(out p);
    Console.WriteLine(p.x.y);
  }
}
```

According to our formalization, the C♯ compiler relies on the set $\mathrm{MFP}_a(\alpha) = \mathit{after}(\alpha) = \{\texttt{p}\}$ with ${}^{\alpha}(\texttt{A.m(out p)})$ when checking the status of the variable `p.x.y`. But, as observed above, after α is evaluated, `p.x` and `p.x.y` are considered assigned as well. So allowing struct type variables requires the compiler to rely on "expanded" sets: the set $\{\texttt{p}\}$ is "expanded" to $\{\texttt{p}, \texttt{p.x}, \texttt{p.x.y}\}$ to include also the instance fields of `p`.
Further, if a local variable loc_2 which is an instance field of a struct type variable loc_1 gets assigned and each instance field of loc_1 except loc_2 either is already considered definitely assigned or gets assigned at the same time as loc_2 (this happens, for example, in case of the `out` arguments following a method call), then the variable loc_1 gets assigned as well and this "lookup" procedure is repeated with loc_1 instead of loc_2. In the following example the struct type field `p.y` gets assigned when its instance field `v` gets assigned. Next, the instance variable `p` gets assigned when its instance field `p.x` gets assigned since `p.y` is already assigned.

```
struct P {
  public int x;
  public Q y;
}

struct Q {
  public int u;
  public int v;
}
```

```
class Test {
  static void Main() {
    P p;
    p.y.u = 1;
    p.y.v = 1;
    p.x = 1;
    P r = p;
  }
}
```

${}^{\alpha}exp$	edges
`true`	$(\mathcal{B}(\alpha), \mathcal{T}(\alpha))$
`false`	$(\mathcal{B}(\alpha), \mathcal{F}(\alpha))$
(`!` ${}^{\beta}e$)	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{F}(\beta), \mathcal{T}(\alpha)), (\mathcal{T}(\beta), \mathcal{F}(\alpha))$
(${}^{\beta}e_0$ `?` ${}^{\gamma}e_1$ `:` ${}^{\delta}e_2$)	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{B}(\delta)),$ $(\mathcal{T}(\gamma), \mathcal{T}(\alpha)), (\mathcal{T}(\delta), \mathcal{T}(\alpha)), (\mathcal{F}(\gamma), \mathcal{F}(\alpha)),$ $(\mathcal{F}(\delta), \mathcal{F}(\alpha))$
(${}^{\beta}e_1$ `&&` ${}^{\gamma}e_2$)	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{F}(\alpha)),$ $(\mathcal{T}(\gamma), \mathcal{T}(\alpha)), (\mathcal{F}(\gamma), \mathcal{F}(\alpha))$
(${}^{\beta}e_1$ `||` ${}^{\gamma}e_2$)	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{T}(\alpha)), (\mathcal{F}(\beta), \mathcal{B}(\gamma)),$ $(\mathcal{T}(\gamma), \mathcal{T}(\alpha)), (\mathcal{F}(\gamma), \mathcal{F}(\alpha))$

Table 4: Control flow for boolean expressions

In accordance with the formalization, the compiler relies on the set $\text{MFP}_a(\alpha) = \mathit{after}(\alpha) = \{\texttt{p.y.u}, \texttt{p.y.v}, \texttt{p.x}\}$ with ${}^{\alpha}(\texttt{p.x} = 1)$ when verifying whether `p` is definitely assigned before evaluating the assignment to `r`. Considering also struct type variables makes the compiler rely on the expanded set of $\{\texttt{p.y.u}, \texttt{p.y.v}, \texttt{p.x}\}$, i.e. $\{\texttt{p.y.u}, \texttt{p.y.v}, \texttt{p.y}, \texttt{p.x}, \texttt{p}\}$.
Hence we conclude that after the C♯ compiler determines the MFP sets, it relies on the expanded MFP sets in order to reject/accept programs. We will say that an "expansion" function is applied to the MFP sets to *propagate* the *definitely assigned* status.

4 THE CONTROL FLOW GRAPH

So far we have seen the equations used for the analysis and we have proven that the fixed point iteration for these equations is well-defined. The main result we want to prove is that the outcome of the analysis is correct: 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* (see [5] for more examples). The nodes of the graph are actually points associated with every expression and statement. We suppose that every expression or statement α is characterized by an *entry point* $\mathcal{B}(\alpha)$ and an *end point* $\mathcal{A}(\alpha)$. Beside these two points a boolean expression α has two more points: a *true point* $\mathcal{T}(\alpha)$ (used when α evaluates to `true`) and a *false point* $\mathcal{F}(\alpha)$ (used when α evaluates to `false`). The edges of

${}^{\alpha}exp$	edges
loc	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
lit	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
$(loc = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$(loc \;\; op = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$({}^{\beta}e_0 \texttt{ ? } {}^{\gamma}e_1 \texttt{ : } {}^{\delta}e_2)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{B}(\delta)),$ $(\mathcal{A}(\gamma), \mathcal{A}(\alpha)), (\mathcal{A}(\delta), \mathcal{A}(\alpha))$
$c.f$	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
$\texttt{ref}\, {}^{\beta}exp$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)),\ (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$\texttt{out}\, {}^{\beta}exp$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)),\ (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$c.m({}^{\beta_1} arg_1, \ldots, {}^{\beta_k} arg_k)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), (\mathcal{A}(\beta_k), \mathcal{A}(\alpha)),$ $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), i = \overline{1, k-1}$

Table 5: Control flow for arbitrary expressions

the graph are given by the *control transfer* defined in the C♯ Specification [1, §8]. Tables 4 and 5 show the edges specific to each boolean and arbitrary expression, respectively. If the expression α is not an instance of one expression in these tables (e.g. $exp_1 \,|\, exp_2$) and has the *direct subexpressions* $\beta_1, \ldots, \beta_n$, then the left-to-right evaluation scheme adds also the following edges to the flow graph:

$$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)),\ (\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})),\ i = \overline{1, n-1} \text{ and } (\mathcal{A}(\beta_n), \mathcal{A}(\alpha))$$

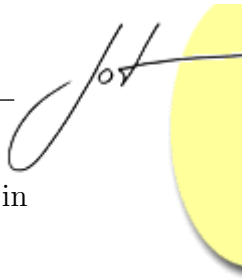
For every boolean expression α in Table 4 we define the supplementary edges $(\mathcal{T}(\alpha), \mathcal{A}(\alpha))$ and $(\mathcal{F}(\alpha), \mathcal{A}(\alpha))$ which connect the boolean points of α to the *end point* of α. These edges are necessary for the control transfer in cases when it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

For a boolean expression α which is not an instance of any expression from Table 4 we add the edges $(\mathcal{A}(\alpha), \mathcal{T}(\alpha))$, $(\mathcal{A}(\alpha), \mathcal{F}(\alpha))$ to the graph. They are needed if control is transferred from a boolean expression α to different points depending on whether α evaluates to `true` or `false`. For example, if α is of the form $exp_1 \,|\, exp_2$ and occurs in ${}^{\beta}\texttt{(!(}exp_1 \,|\, exp_2\texttt{))}$, then control is transferred from $\mathcal{F}(\alpha)$ to $\mathcal{T}(\beta)$ (if α evaluates to `false`) or from $\mathcal{T}(\alpha)$ to $\mathcal{F}(\beta)$ (if α evaluates to `true`). The necessity of the edges $(\mathcal{A}(\alpha), \mathcal{T}(\alpha))$, $(\mathcal{A}(\alpha), \mathcal{F}(\alpha))$ arises since so far we have defined for $exp_1 \,|\, exp_2$ only edges to $\mathcal{A}(\alpha)$.

${}^{\alpha}stm$	edges
`;`	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
$({}^{\beta}exp\,\texttt{;})$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$\{{}^{\beta_1}stm_1 \ldots {}^{\beta_n}stm_n\}$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), (\mathcal{A}(\beta_n), \mathcal{A}(\alpha))$, $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), i = \overline{1, n-1}$
`if` $({}^{\beta}exp)\ {}^{\gamma}stm_1$ `else` ${}^{\delta}stm_2$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{B}(\delta))$, $(\mathcal{A}(\gamma), \mathcal{A}(\alpha)), (\mathcal{A}(\delta), \mathcal{A}(\alpha))$
`while` $({}^{\beta}exp)\ {}^{\gamma}stm$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)), (\mathcal{F}(\beta), \mathcal{A}(\alpha))$, $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$
`L:` ${}^{\beta}stm$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
`goto L;`	$ThroughFin_b(\alpha, \beta)$, where ${}^{\beta}$`L:`stm is the statement to which α points
`break;`	$ThroughFin_a(\alpha, \beta)$, where β is the nearest enclosing `while` with respect to α
`continue;`	$ThroughFin_b(\alpha, \beta)$, where β is the nearest enclosing `while` with respect to α
`return;`	no edges
`return` ${}^{\beta}exp$`;`	$(\mathcal{B}(\alpha), \mathcal{B}(\beta))$
`throw;`	no edges
`throw` ${}^{\beta}exp$`;`	$(\mathcal{B}(\alpha), \mathcal{B}(\beta))$
`try` ${}^{\beta}block$ `catch(`$E_1\ x_1$`)` ${}^{\gamma_1}block_1$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
⋮	$(\mathcal{B}(\alpha), \mathcal{B}(\gamma_i)), (\mathcal{A}(\gamma_i), \mathcal{A}(\alpha))\ i = \overline{1, n}$
`catch(`$E_n\ x_n$`)` ${}^{\gamma_n}block_n$	
`try` ${}^{\beta}block_1$ `finally` ${}^{\gamma}block_2$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)),(\mathcal{B}(\alpha), \mathcal{B}(\gamma)), (\mathcal{A}(\beta), \mathcal{B}(\gamma))$ and $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ conditioned by $\mathcal{A}(\beta)$

Table 6: Control flow for statements

Table 6 introduces the edges of the control flow graph for each statement. We assume that the boolean constant expressions are replaced by `true` or `false` in the



abstract syntax tree. For example, we consider that `true||b` is replaced by `true` in the following `if` statement:

```
α if β(true||b) δi = 1;
  else γ{ int j = i; }
```

Although the new test (i.e. `true`) cannot evaluate to `false`, we still add to the graph the edge $(\mathcal{F}(\beta), \mathcal{B}(\gamma))$: however the *false point* of `true` is never reachable (see Table 4).

In the presence of `finally` blocks the jump statements `goto`, `continue` and `break` bring more complexity to the graph. When a jump statement exits a `try` block, control is transferred first to the innermost `finally` block. If control reaches the *end point* of that `finally` block, then it is transferred to the next innermost `finally` block and so on. If control reaches the *end point* of the outermost `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 of some important details: 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 *conditioned* 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 $[\mathcal{B}(mb)\mathcal{B}(\alpha_1)\mathcal{B}(\alpha_2)\mathcal{B}(\alpha_3)\mathcal{B}(\alpha_4)\mathcal{B}(\alpha_5)\mathcal{A}(\alpha_5)\mathcal{B}(\alpha_6)]$ would be a possible execution path to the labeled statement in the following method body

```
α1try α2 {
    α3( α4(i = 3/j);)
     αgoto L;
   } finally α5{}
α6L:Console.WriteLine(i);
```

in case the evaluation of α_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 α and β are two statements and $Fin(\alpha, \beta)$ is the list $[\gamma_1, \ldots, \gamma_n]$, then the set $ThroughFin_b(\alpha, \beta)$ consists of the edges $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_1))$, $(\mathcal{A}(\gamma_n), \mathcal{B}(\beta))$, $(\mathcal{A}(\gamma_i), \mathcal{B}(\gamma_{i+1}))$, $i = \overline{1, n-1}$, all conditioned by $\mathcal{B}(\alpha)$, while the set $ThroughFin_a(\alpha, \beta)$ contains the edges $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_1))$, $(\mathcal{A}(\gamma_n), \mathcal{A}(\beta))$, $(\mathcal{A}(\gamma_i), \mathcal{B}(\gamma_{i+1}))$, $i = \overline{1, n-1}$ all conditioned by $\mathcal{B}(\alpha)$. If $Fin(\alpha, \beta)$ is empty, then the set $ThroughFin_b(\alpha, \beta)$ contains only the edge $(\mathcal{B}(\alpha), \mathcal{B}(\beta))$ while $ThroughFin_a(\alpha, \beta)$ refers to the edge $(\mathcal{B}(\alpha), \mathcal{A}(\beta))$. In the previous example the list $Fin(\alpha, \alpha_6)$ is given by $[\alpha_5]$ while the set $ThroughFin_b(\alpha, \alpha_6)$ contains the edges $(\mathcal{B}(\alpha), \mathcal{B}(\alpha_5))$, $(\mathcal{A}(\alpha_5), \mathcal{B}(\alpha_6))$ conditioned by $\mathcal{B}(\alpha)$.

Note that in Table 6, for `goto` and `continue`, the set of edges $ThroughFin_b$ is added to the graph, since after executing the `finally` blocks control is transferred to the *entry point* of the labeled statement and `while` statement, respectively. However, in case of `break` the set $ThroughFin_a$ is considered since at the end control is

transferred to the *end point* of the `while` statement.

There are two more remarks concerning the `try` statement. First of all, since a reason for abruption (e.g. an exception) can occur anytime in a `try` block, we should have edges from every point in a `try` block to: every associated `catch` block, every `catch` of enclosing `try` statements (if the `catch` clause matches the type of the exception) and to every associated `finally` block (if none of the `catch` clauses matches the type of the exception). We do not consider all these edges since the definite assignment analysis is an "over all paths" analysis. It is equivalent to consider only one edge to the *entry point*s of the `catch` and `finally` blocks — from the *entry point* of the `try` block (see Table 6).

The next remark concerns the *end point* $\mathcal{A}(\alpha)$ of a `try-finally` statement α. The C♯ Specification states in [§8.10] that $\mathcal{A}(\alpha)$ is reachable only if both *end point*s of the `try` block β and `finally` block γ 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 the case of a jump, if control reaches the *end point* $\mathcal{A}(\gamma)$ of the `finally`, then it is transferred further to the target of the statement which generated the jump and not to $\mathcal{A}(\alpha)$. This means that all paths to $\mathcal{A}(\alpha)$ contain also the *end point* $\mathcal{A}(\beta)$ of the `try` block. That is why we require that the edge $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ is *conditioned* by $\mathcal{A}(\beta)$ (see Table 6) — otherwise in the following example $\mathcal{A}(\alpha)$ would be reachable in our graph (under the assumption that $\mathcal{B}(\alpha)$ is reachable):

$^{\alpha}$ `try` $^{\beta}$`{ goto L; }` `finally` $^{\gamma}$ `{}`

Therefore 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. Formally:

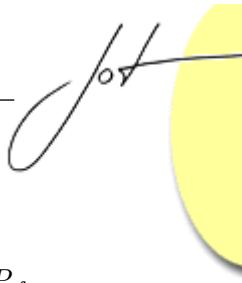
$$\begin{aligned} valid([\alpha_1, \ldots, \alpha_n]) \quad \equiv \quad & \text{for every } \textit{conditioned} \text{ edge } (\alpha_i, \alpha_{i+1}), \\ & \exists\, j < i \text{ such that } (\alpha_i, \alpha_{i+1}) \text{ is } \textit{conditioned} \text{ by } \alpha_j \end{aligned}$$

If α is an expression or a statement, then $path_b(\alpha)$ is the set of all valid paths from the *entry point* of the method body $\mathcal{B}(mb)$ to the *entry point* $\mathcal{B}(\alpha)$ of α:

$$\begin{aligned} path_b(\alpha) \quad = \quad & \{[\alpha_1, \ldots, \alpha_n] \mid \alpha_1 = \mathcal{B}(mb), \alpha_n = \mathcal{B}(\alpha), (\alpha_i, \alpha_{i+1}) \in CFG, i = \overline{1, n-1} \\ & \text{and } valid([\alpha_1, \ldots, \alpha_n])\} \end{aligned}$$

Similarly $path_a(\alpha)$ is the set of all valid paths from the *entry point* of the method body $\mathcal{B}(mb)$ to the *end point* $\mathcal{A}(\alpha)$ of α, while if α is a boolean expression, $path_t(\alpha)$ and $path_f(\alpha)$ are the sets of all valid paths from $\mathcal{B}(mb)$ to the *true point* $\mathcal{T}(\alpha)$ and to the *false point* $\mathcal{F}(\alpha)$ of α, respectively.

In the proofs in the next section we 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 consider the operation $\oplus$ to be path concatenation (defined also for infinite paths).



5 CORRECTNESS OF THE ANALYSIS

We prove that when a C♯ compiler relies on the sets MFP_b, MFP_a, MFP_t and MFP_f derived from the maximal fixed point of the data flow equations in Section 2 (or on their expanded sets if we allow struct type variables), then all accesses to the value of a local variable occur after it is initialized. 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 path of the program. A variable *loc* is *assigned* on a path if the path contains an *initialization* of *loc* or a `catch` clause whose exception variable is *loc*. We describe in the following definition what we mean by *initialization*.

Definition 2 *A path p contains an initialization of a local variable loc if at least one of the following is true:*

- *p contains a simple assignment (not a compound assignment) to loc, or*
- *p contains a method invocation for which loc is an* `out` *argument.*

Struct type variables. The definition above has to be extended if we also want to allow variables of struct types. Thus a path p contains an *initialization* of a local variable *loc* **also** in one of the following cases:

- *loc* is an instance field of a struct type variable x and p contains an *initialization* of x, or
- *loc* is of a struct type and p contains *initializations* for each instance field of *loc*.

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

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

$\mathrm{AP}_a(\alpha)$ is the set of variables in $vars(\alpha)$ which are assigned on every path in $path_a(\alpha)$, while for a boolean expression α the sets $\mathrm{AP}_t(\alpha)$ and $\mathrm{AP}_f(\alpha)$ are defined similarly as above, but with respect to paths in $path_t(\alpha)$ and $path_f(\alpha)$, respectively.

Struct type variables. If we consider also variables of struct types, the definition of "is *assigned* on" is extended as pointed out above. The definitions of the sets AP_b, AP_a, AP_t and AP_f are also adapted. But considering the new definition for "is *assigned*", one can easily observe that the definitions of the AP sets to include struct type variables are nothing else than their expanded sets. So actually the same "expansion" function we mentioned in Section 3 is applied also to the AP sets in order to include struct type variables.

The following result is used to prove Lemma 5.

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

Proof. The proof proceeds by induction over the structure of expressions and statements. Thus, we first prove the base cases of the induction, i.e. the above stated implications for all possible leaves of the abstract syntax tree (AST) of our method body. The expressions which are leaves in the AST are the following: `true`, `false`, *loc*, *lit* and *c.f*. Since MFP is in particular a solution of the data flow equations, it is obvious that the implications stated in our lemma are satisfied. The statements considered leaves in the AST are the *empty-statement*, `goto L`, `break`, `continue`, `return` and `throw`. For the last five, from the equations above we obviously have $\mathrm{MFP}_a(\alpha) \subseteq vars(\alpha)$. For the *empty-statement* this is true as well since our hypothesis is $\mathrm{MFP}_b(\alpha) \subseteq vars(\alpha)$.
In the induction step the implications for each expression and statement are proved under the assumption that their "children" (subexpressions/substatements) satisfy the implications. □

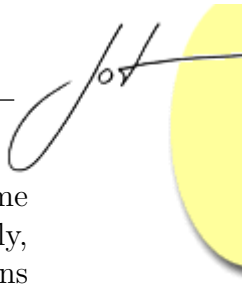
The next lemma is used in the proof of the correctness theorem (Theorem 1). It claims that the MFP sets of an expression or statement α consist of variables in the scope of which α is located.

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

Proof. We show the above inclusions for all expressions and statements by an induction over the AST, starting at the root, i.e. the method body (the basis of induction). Notice that the induction schema is in the opposite direction compared to that in Lemma 4. Therefore the induction step is: under the assumption that a node of the AST satisfies the inclusions, all its "children" (subexpressions/substatements) satisfy the inclusions as well.
According to Lemma 4 it is enough to prove for all labels α: $\mathrm{MFP}_b(\alpha) \subseteq vars(\alpha)$. For our method body this is trivial: the relation $\mathrm{MFP}_b(mb) \subseteq vars(mb)$ holds since $\mathrm{MFP}_b(mb) = vars(mb) = \emptyset$. Lemma 4 is used again in the next step of the proof, which consists in showing for each expression and statement that under the assumption $\mathrm{MFP}_b(\alpha) \subseteq vars(\alpha)$, each of its direct subexpressions/substatements β satisfies $\mathrm{MFP}_b(\beta) \subseteq vars(\beta)$. □

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 α, the following relations are true: $\mathrm{MFP}_b(\alpha) \subseteq \mathrm{AP}_b(\alpha)$ and $\mathrm{MFP}_a(\alpha) \subseteq \mathrm{AP}_a(\alpha)$. Moreover, if α is a boolean expression, then we have also $\mathrm{MFP}_t(\alpha) \subseteq \mathrm{AP}_t(\alpha)$ and $\mathrm{MFP}_f(\alpha) \subseteq \mathrm{AP}_f(\alpha)$.*



Proof. We consider the following definitions. The set $\mathrm{AP}_b^n(\alpha)$ is defined in the same way as $\mathrm{AP}_b(\alpha)$, except that we consider only paths of length less than n. Similarly, we also define the sets $\mathrm{AP}_a^n(\alpha)$, $\mathrm{AP}_t^n(\alpha)$, $\mathrm{AP}_f^n(\alpha)$ (analogously, we have definitions for the sets of paths $path_b^n$, $path_a^n$, $path_t^n$, $path_f^n$). According to these definitions, the following set equalities hold for every α:

$$\mathrm{AP}_b(\alpha) = \bigcap_n \mathrm{AP}_b^n(\alpha),\ \ \mathrm{AP}_a(\alpha) = \bigcap_n \mathrm{AP}_a^n(\alpha)$$

If α is a boolean expression, then similar equalities hold for $\mathrm{AP}_t(\alpha)$ and $\mathrm{AP}_f(\alpha)$.

Therefore, to complete the proof it suffices to show for every n: if α is an expression or statement, then $\mathrm{MFP}_b(\alpha) \subseteq \mathrm{AP}_b^n(\alpha)$ and $\mathrm{MFP}_a(\alpha) \subseteq \mathrm{AP}_a^n(\alpha)$. In addition, if α is a boolean expression, then $\mathrm{MFP}_t(\alpha) \subseteq \mathrm{AP}_t^n(\alpha)$ and $\mathrm{MFP}_f(\alpha) \subseteq \mathrm{AP}_f^n(\alpha)$. This is done by induction on n.

Basis of induction. $[\mathcal{B}(mb)]$ 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 $\mathrm{AP}_b^1(mb) = \emptyset$ which satisfies $\mathrm{MFP}_b(mb) \subseteq \mathrm{AP}_b^1(mb)$ since from the equations $\mathrm{MFP}_b(mb) = \emptyset$. From the definition of AP_a^1, we get $\mathrm{AP}_a^1(mb) = vars(mb) = \emptyset$ and from the equations of a block, we derive also $\mathrm{MFP}_a(mb) \subseteq vars(mb)$ and implicitly $\mathrm{MFP}_a(mb) \subseteq \mathrm{AP}_a^1(mb)$. If $\alpha \neq mb$, then $\mathrm{AP}_b^1(\alpha) = \mathrm{AP}_a^1(\alpha) = vars(\alpha)$ and if α is a boolean expression $\mathrm{AP}_t^1(\alpha) = \mathrm{AP}_f^1(\alpha) = vars(\alpha)$. If we apply Lemma 5, then the basis of the induction is complete.

Induction step. The proof has the following pattern: we show for every expression or statement α from Tables 1, 2, and 3 the relation for $\mathrm{MFP}_a(\alpha)$, and where applicable for $\mathrm{MFP}_t(\alpha)$ and $\mathrm{MFP}_f(\alpha)$ and, for every direct subexpression/substatement of α, the relations for MFP_b. In this way, all the relations for all expressions/statements are proved except $\mathrm{MFP}_b(mb) \subseteq \mathrm{AP}_b^{n+1}(mb)$ (since mb has no "superstatement") which holds anyway since $\mathrm{MFP}_b(mb) = \emptyset$.

We consider here only two critical cases (see [5] for the complete proof).

Case 1 *block of statements*

Let us consider the case when α is a block of statements: $\{^{\beta_1} stm_1 \ldots {}^{\beta_n} stm_n\}$.

We prove $\mathrm{MFP}_b(\beta_{i+1}) \subseteq \mathrm{AP}_b^{n+1}(\beta_{i+1})$ for an embedded statement β_{i+1}. If we arbitrarily choose a local variable x in $\mathrm{MFP}_b(\beta_{i+1})$, then we obtain $x \in \mathrm{MFP}_a(\beta_i)$ and $x \in goto(\beta_{i+1})$ (MFP is a solution of the flow equations). Note that, at this point, $goto(\beta_{i+1})$ depends only on MFP sets and on the control flow graph. From the induction hypothesis, we get $x \in \mathrm{AP}_a^n(\beta_i)$. In particular, this means $x \in vars(\beta_i) = vars(\beta_{i+1})$, i.e. β_{i+1} is in the scope of a declaration of x.

Case 1.1 β_{i+1} *is not a labeled statement*

Let us suppose that β_{i+1} is not a labeled statement. In this situation we have $goto(\beta_{i+1}) = vars(\beta_{i+1})$ from the definition of the *goto* set.

Case 1.1.1 β_{i+1} *is not a* `while` *statement*

If additionally β_{i+1} is not a `while` statement, then there exists in the *CFG* only one edge to $\mathcal{B}(\beta_{i+1})$, namely $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$. This means

$$path_b^{n+1}(\beta_{i+1}) = path_a^n(\beta_i) \oplus \mathcal{B}(\beta_{i+1})$$

and implicitly $x \in \mathrm{AP}_b^{n+1}(\beta_{i+1})$ since $x \in \mathrm{AP}_a^n(\beta_i)$ (induction hypothesis). Remember that we derived earlier that β_{i+1} is in the scope of x, i.e. $x \in vars(\beta_{i+1})$.

Case 1.1.2 *β_{i+1} is a* `while` *statement*

If β_{i+1} is a `while` statement, then there could be many edges to $\mathcal{B}(\beta_{i+1})$ in the *CFG* (if the `while` has associated `continue` statements). If there are no `continue` statements corresponding to our `while` statement, then the proof is as in **Case 1.1.1**.

If there are `continue` statements, we would like to show that x is assigned on every path p to $\mathcal{B}(\beta_{i+1})$ of length at most $n+1$. If p contains no `continue` statements, then necessarily p has the last edge $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$, i.e. p passes through $\mathcal{A}(\beta_i)$. If p contains a `continue` statement associated to our `while` and eventually passes through `finally` blocks associated to `try` blocks that contain the `continue` statement, then necessarily p passes through $\mathcal{A}(\beta_i)$ because the *CFG* shows that it is not possible to "jump" into the `while` body (in which our `continue` is embedded). On the other hand, since $x \in \mathrm{MFP}_a(\beta_i)$, we get $x \in \mathrm{AP}_a^n(\beta_i)$ from the induction hypothesis, i.e. x is assigned on every path to $\mathcal{A}(\beta_i)$ of length at most n. Consequently, p assigns x since p passes through $\mathcal{A}(\beta_i)$. Now we are sure that x is assigned on every path to $\mathcal{B}(\beta_{i+1})$ of length at most $n+1$ and therefore we can conclude $x \in \mathrm{AP}_b^{n+1}(\beta_{i+1})$.

Case 1.2 *β_{i+1} is a labeled statement*

Let us suppose now that β_{i+1} is a labeled statement `L:`*stm*. Then, as in the case of a `while` statement, there could be many edges to $\mathcal{B}(\beta_{i+1})$ in the *CFG* (if there are `goto` statements pointing to our labeled statement). If there are no associated `goto` statements, then the proof is the same as in the case of a `while` statement with no associated `continue` statements.

We want to show that x is assigned on every path p to $\mathcal{B}(\beta_{i+1})$ of length at most $n+1$. If p contains no `goto L` statements, then necessarily p passes through $\mathcal{A}(\beta_i)$. And since x is assigned on every path to $\mathcal{A}(\beta_i)$ of length at most n (because of $x \in \mathrm{AP}_a^n(\beta_i)$), we are sure p assigns x.

Suppose that p passes through a $^{\gamma}$`goto L` statement and eventually through `finally` blocks associated to `try` blocks in which γ is embedded. Considering the definition of the *goto* set, we can derive $x \in \mathrm{MFP}_b(\gamma) \cup \mathit{JoinFin}(\gamma, \beta_{i+1})$ since $x \in \mathit{goto}(\beta_{i+1})$.

If there are no `finally` blocks in the list $\mathit{Fin}(\gamma, \beta_{i+1})$, then $\mathit{JoinFin}(\gamma, \beta_{i+1}) = \emptyset$ and implicitly $x \in \mathrm{MFP}_b(\gamma)$. Using the induction hypothesis, we obtain $x \in \mathrm{AP}_b^n(\gamma)$. It means that p, which is of length at most $n+1$ and contains $\mathcal{B}(\gamma)$, assigns x.

Let us suppose now that the list $\mathit{Fin}(\gamma, \beta_{i+1})$ is non-empty: $\mathit{Fin} = [\gamma_1, \ldots, \gamma_k]$. From the definition of the set $\mathit{JoinFin}(\gamma, \beta_{i+1})$ we get

$$x \in \mathrm{MFP}_b(\gamma) \cup \bigcup_{j=1}^{k} \mathrm{MFP}_a(\gamma_j)$$

If $x \in \mathrm{MFP}_b(\gamma)$, then from the induction hypothesis we derive $x \in \mathrm{AP}_b^n(\gamma)$ and we are sure that p, which passes through $\mathcal{B}(\gamma)$, assigns x.

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

Thus, we have 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, i.e $x \in \mathrm{AP}_b^{n+1}(\beta_{i+1})$.

Case 2 `try-finally` *statement*
Let us consider now the case where α is of the form `try` $^{\beta}$*block*$_1$ `finally` $^{\gamma}$*block*$_2$.
Here we study the proof for $\mathrm{MFP}_b(\gamma) \subseteq \mathrm{AP}_b^{n+1}(\gamma)$. Let x be a local variable in the set $\mathrm{MFP}_b(\gamma)$. Following the data flow equations, we get $x \in \mathrm{MFP}_b(\alpha)$. The induction hypothesis implies then $x \in \mathrm{AP}_b^n(\alpha)$ and in particular $x \in \mathit{vars}(\alpha)$.
It is important to notice that in the *CFG* there could be many edges to $\mathcal{B}(\gamma)$: from the *entry point* of the `try-finally` statement $(\mathcal{B}(\alpha), \mathcal{B}(\gamma))$, from the *end point* of the `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), and from the *end point* of another `finally` block $(\mathcal{A}(\omega), \mathcal{B}(\gamma))$ (within a *conditioned* path). We claim that independent of the last edge of a path p to $\mathcal{B}(\gamma)$, p passes through the *entry point* $\mathcal{B}(\alpha)$ of the `try-finally` statement.

- If the last edge of p is $(\mathcal{B}(\alpha), \mathcal{B}(\gamma))$, then there is nothing to prove.
- If the last edge is $(\mathcal{A}(\beta), \mathcal{B}(\gamma))$, then the claim holds since *the end point* $\mathcal{A}(\beta)$ can be reached only through $\mathcal{B}(\alpha)$ (according to the *CFG*, it is not possible to "jump" into the `try` block).
- If the last edge is $(\mathcal{B}(\delta), \mathcal{B}(\gamma))$, then the claim can be justified in the same way as 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 ω (as well as the jump statement which triggered the conditioning) is embedded in our `try` block. This means that in order to justify the claim we can apply the same argument as above.

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

As explained above we can actually prove more: the *MFP* solution is not only an approximation of *AP* but it is a perfect solution (Theorem 3). For this, we also use the following theorem that states that the *MFP* solution contains the local variables which are initialized over *all possible paths*.

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

Proof. Tarski's fixed point theorem [13] states that *MFP* is the lowest upper bound (with respect to $\sqsubseteq$) of the set $\mathit{Ext}(F) = \{X \in D \mid X \sqsubseteq F(X)\}$. It then suffices to show that the r-tuple consisting of the *AP* sets is an element of $\mathit{Ext}(F)$ since *MFP* is in particular an upper bound of this set. Since $\sqsubseteq$ is the pointwise subset relation, the idea is to prove the left-to-right subset relations for the data flow equations in

Tables 1, 2, and 3, where instead of the sets *before*, *after*, *true* and *false* we have the sets AP_b, AP_a AP_t and AP_f, respectively.

Here we consider only one critical case, encountered for a block of statements (see [5] for the complete proof): assuming that β_{i+1} is a labeled statement `L:` stm in a block α given by $\{^{\beta_1} stm_1 \ldots {}^{\beta_n} stm_n\}$, we want to prove $\mathrm{AP}_b(\beta_{i+1}) \subseteq \mathrm{AP}_a(\beta_i) \cap goto(\beta_{i+1})$. Note that here, $goto(\beta_{i+1})$ depends only on the AP sets and on the control flow graph CFG.

Let x be a variable in $\mathrm{AP}_b(\beta_{i+1})$, i.e. x is assigned on every path to $\mathcal{B}(\beta_{i+1})$. An immediate consequence is $x \in \mathrm{AP}_a(\beta_i)$ since all the paths to $\mathcal{A}(\beta_i)$ are — "modulo" the edge $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$ — also paths to $\mathcal{B}(\beta_{i+1})$ and no variable is assigned on this edge.

In order to show $x \in goto(\beta_{i+1})$ we need to prove that the variable x is in the set $\mathrm{AP}_b(\gamma) \cup JoinFin(\gamma, \beta_{i+1})$ for every $^{\gamma}$`goto L` whose target is our labeled statement.

If there is no such `goto` statement, then we obviously have $x \in goto(\beta_{i+1})$ since in this case $goto(\beta_{i+1}) = vars(\beta_{i+1})$ and $x \in \mathrm{AP}_b(\beta_{i+1}) \subseteq vars(\beta_{i+1})$. Let us suppose now there exists at least one `goto` statement γ pointing to β_{i+1}.

Case 1 $\mathcal{B}(\gamma)$ *not reachable*

If $\mathcal{B}(\gamma)$ is not reachable in the CFG, then $path_b(\gamma)$ is empty and consequently we get $x \in \mathrm{AP}_b(\gamma) \cup JoinFin(\gamma, \beta_{i+1})$ because $\mathrm{AP}_b(\gamma) = vars(\gamma) \supseteq vars(\beta_{i+1})$ and $x \in vars(\beta_{i+1})$. The last subset relation holds because, in C♯, a `goto` statement should be always in the scope of the corresponding labeled statement.

Case 2 $\mathcal{B}(\gamma)$ *reachable*

If $\mathcal{B}(\gamma)$ is reachable in the CFG, then let p be an arbitrary path to $\mathcal{B}(\gamma)$. Here we will only consider the case there are `finally` blocks from γ to β_{i+1}, i.e. $Fin(\gamma, \beta_{i+1}) = [\gamma_1, \ldots, \gamma_k]$ (the proof of the case where there are no `finally` blocks is much simpler and we refer the interested reader to [5]). Accordingly, also the edges

$$(\mathcal{B}(\gamma), \mathcal{B}(\gamma_1)), (\mathcal{A}(\gamma_k), \mathcal{B}(\beta_{i+1})), (\mathcal{A}(\gamma_j), \mathcal{B}(\gamma_{j+1})), j = \overline{1, k-1}$$

defined by the set $ThroughFin_b(\gamma, \beta_{i+1})$ are added to the CFG.

We will prove $x \in \mathrm{AP}_b(\gamma) \cup JoinFin(\gamma, \beta_{i+1})$ by contradiction. Let us assume that $x \notin \mathrm{AP}_b(\gamma) \cup JoinFin(\gamma, \beta_{i+1})$. This is equivalent to $x \notin \mathrm{AP}_b(\gamma)$ and $x \notin \mathrm{AP}_a(\gamma_j)$ for all $j = \overline{1, k}$. This means that the paths $p_0 \in path_b(\gamma)$ and $p_j \in path_a(\gamma_j)$ for $j = \overline{1, k}$ exist, such that x is not assigned on any of these paths. A simple inspection of the CFG shows that the point $\mathcal{B}(\gamma_j)$ necessarily occurs on the path p_j for every $j = \overline{1, k}$ since it is not possible to "jump" into a `finally` block. We want to prove now that the following list

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

represents a valid path to $\mathcal{B}(\beta_{i+1})$. The only problem that could arise is concerning the *conditioned* edges. Remember that the edges *conditioned* by a certain `goto`, `break` or `continue` statement can be used only in paths (or subpaths) that contain the *entry point* of the respective jump statement. The use of edges $(\mathcal{B}(\gamma), \mathcal{B}(\gamma_1)), \ldots (\mathcal{A}(\gamma_k), \mathcal{B}(\beta_{i+1}))$ is correct as long as our path q contains $\mathcal{B}(\gamma)$.

Let us consider one of the subpaths $p_j[\mathcal{B}(\gamma_j), \mathcal{A}(\gamma_j)]$ used in q. If this subpath contains a *conditioned edge*, then since the *conditioned* edges connect jump statements with `finally` blocks we would be sure that these `finally` blocks are embedded in our `finally` block γ_j. The respective jump statement is embedded into the `try` blocks (associated to the *conditioned* "connected" `finally` blocks) which necessarily should be in γ_j (this is an immediate consequence of the C♯ grammar). So the jump statement is necessarily embedded in the `finally` block γ_j. Considering that the subpath $p_j[\mathcal{B}(\gamma_j), \mathcal{A}(\gamma_j)]$ 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 $p_j[\mathcal{B}(\gamma_j), \mathcal{A}(\gamma_j)]$. It means that this subpath is valid. Obviously, this is true for all the considered subpaths in q.

The above defined path q is a valid path to $\mathcal{B}(\beta_{i+1})$ which does not assign x. Obviously, this contradicts $x \in \mathrm{AP}_b(\beta_{i+1})$ and therefore our assumption is wrong. Hence, we obtain the desired $x \in \mathrm{AP}_b(\gamma) \cup \mathit{JoinFin}(\gamma, \beta_{i+1})$. □

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, and 3 represents the sets of local variables which are assigned over all possible execution paths. More exactly, for every expression or statement α, the following are true:* $\mathrm{AP}_b(\alpha) = \mathrm{MFP}_b(\alpha)$ *and* $\mathrm{AP}_a(\alpha) = \mathrm{MFP}_a(\alpha)$. *Moreover, if α is a boolean expression:* $\mathrm{AP}_t(\alpha) = \mathrm{MFP}_t(\alpha)$ *and* $\mathrm{AP}_f(\alpha) = \mathrm{MFP}_f(\alpha)$.

Struct type variables. Suppose now that we include for the analysis also variables of struct types. In this case, the C♯ compiler relies on the expanded MFP sets and the correctness of the analysis would mean that the expanded MFP sets are a *safe approximation* of the expanded AP sets. On the other hand, we proved in Theorem 3, that the MFP and AP sets coincide. Then, also the expanded MFP sets will coincide with the expanded AP sets; so they are, in particular, a *safe approximation*. This means that, allowing variables of struct types does not affect the correctness of the definite assignment analysis. One can justify the correctness also by applying Theorem 1 and observing that the "expansion" function is monotonic.

6 CONCLUSION

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 always exists a maximal fixed point solution MFP. We showed the correctness of the analysis, i.e. MFP is a *safe approximation* of the sets of variables assigned over all possible paths. This is a key property for the type safety of C♯. The formalization of the type safety is future work as well as proving the correctness of C♯ compilers.

Our formalization cannot be done as in Java (see [6, 7]), because as we have seen, certain key aspects of C♯ (e.g. `goto`, `ref`/`out`, structs) are not present in Java (this makes the analysis simpler in Java). Moreover, because of the `goto` statement, solving the analysis in C♯ requires a fixed point iteration. Therefore, the type system approach of Schirmer [7] cannot be applied for C♯.

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 [11]. During the attempts to build this model a few discrepancies between the C♯ Specification and different implementations of C♯ were discovered [12].

Acknowledgment I would like to thank the anonymous reviewers for their valuable comments that helped to improve this paper.

References

[1] S. Wiltamuth and A. Hejlsberg. C♯ Language Specification. MSDN, 2003.

[2] Microsoft .NET Framework 1.1 Software Development Kit. Download from http://msdn.microsoft.com/netframework/howtoget/.

[3] Rotor–Shared Source Common Language Infrastructure (SSCLI). Web pages at http://msdn.microsoft.com/net/sscli/ and http://www.sscli.net/.

[4] The Mono project. Web pages at http://www.go-mono.com/.

[5] N. G. Fruja. The Correctness of the Definite Assignment Analysis in C♯. Technical Report 435, ETH Zürich, 2004. http://www.inf.ethz.ch/~fruja.

[6] R. F. Stärk, J. Schmid, E. Börger. Java and the Java Virtual Machine–Definition, Verification, Validation. Springer–Verlag, 2001.

[7] N. Schirmer. Java Definite Assignment in Isabelle/HOL. Proceedings of the Workshop on Formal Techniques for Java–like Programs (*FTfJP'03*), Germany, pages 13-21, 2003.

[8] F. Nielson, H.R. Nielson, C. Hankin. Principles of Program Analysis. Springer–Verlag, 1999.

[9] D. Grune, H. Bal, C. Jacobs, K. Langendoen. Modern Compiler Design. Wiley, 2000.

[10] J. Gough. Compiling for the .NET. Common Language Runtime (CLR). Prentice Hall, 2002.

[11] E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C♯. To appear in Journal Theoretical Computer Science, 2004.

[12] N. G. Fruja. Specification and Implementation Problems for C♯. Proceedings of the Workshop on Abstract State Machines (*ASM'04*), Germany, pages 127-143, 2004.

[13] A. Tarski. A lattice–theoretical fix–point theorem and its applications. Pacific Journal of Mathematics 5, 1955.

ABOUT THE AUTHORS

Nicu G. Fruja is a PhD student and research assistant at the Computer Science Department at the ETH Zürich, Switzerland. His research interests are in the areas of formal methods, specification, verification and validation of systems, abstract state machines, semantics of programming languages, bytecode verification. He can be reached at fruja@inf.ethz.ch. See also http://www.inf.ethz.ch/personal/fruja/.