The correctness of the definite assignment analysis in C#

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

ABSTRACT

In C# the compiler guarantees that each local variable is initialized before an access to its value occurs at runtime. This prevents access to uninitialized memory and is a crucial ingredient for the type safety of C#. We formalize the definite assignment analysis of the C# compiler with data flow equations and we prove the correctness of the analysis.

Keywords
definite assignment, C#, type safety, static analysis

1 INTRODUCTION

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

```
{
  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 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 [Nie99, Gru00] 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 [Wil03, §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. [Sta01] 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 [Sch03]. 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 C# definite assignment analysis we provide, sheds some light in particular on the complications generated by the `goto` and `break` statements (incompletely specified in [Wil03]) 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 [Sta01]) 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*.

A series of bugs in the Mono C# compiler were detected during the attempt to build the formalization of the definite assignment analysis (see [Fru03b] for details). This is the reason we refer here only to .NET and Rotor C# compilers. A bug in the assignment analysis of the Rotor C# compiler is mentioned also here.

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. Due to space limitations we do not make here the proofs in detail. We focus on illustrating how we deal with the jump statements and their complications in the presence of `finally` blocks. The full details, as well as further examples can be found in the full technical report [Fru03b].

2 THE DATA FLOW EQUATIONS

In this section, we formalize the rules of definite assignment analysis from the C# Specification [Wil03, §5.3] by data flow equations. Since this 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

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

Table 1: Definite assignment for boolean expressions

${}^{\alpha}stm$. 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 α and have the following meanings. $before(\alpha)$ contains the local variables definitely assigned before the evaluation of α and $after(\alpha)$ the variables definitely assigned after the evaluation of α when α completes normally. $true(\alpha)$ and $false(\alpha)$ consist of the variables definitely assigned after the evaluation of α when α evaluates to `true` and `false`, respectively. $vars(\alpha)$ contains the local variables in the scope of which α is.

We skip those language constructs (e.g. `foreach`, `for`, `do`, `switch`, `++`, `--`) whose analysis is similar to the one of the constructs dealt with explicitly in our framework. Note that the definite assignment analysis for variables of a struct type is a little bit different: such a local variable is considered definitely assigned iff all its instance fields are definitely assigned. The structs are not considered here but their detailed analysis is included in the full technical report [Fru03b].

A equation is given by initial conditions: for the method body *mb* of *meth* we have $before(mb) = \emptyset$. Actually we should consider the set of value and reference parameters of *meth* but there is no worry that an access to any of them could cause troubles since when *meth* is invoked they are supposed to be definitely assigned [Wil03, §5.1].

For the other expressions and statements in *mb*, instead of explaining how the functions are computed,

$^{\alpha}\mathit{exp}$	the data flow equations
loc	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
lit	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
$(\mathit{loc} = {}^{\beta}e)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta) \cup \{\mathit{loc}\}$
$(\mathit{loc}\,\mathit{op} = {}^{\beta}e)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
$({}^{\beta}e_0\ ?\ {}^{\gamma}e_1\ :\ {}^{\delta}e_2)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{true}(\beta)$ $\mathit{before}(\delta) = \mathit{false}(\beta)$ $\mathit{after}(\alpha) = \mathit{after}(\gamma) \cap \mathit{after}(\delta)$
$c.f$	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
`ref` $^{\beta}\mathit{exp}$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
`out` $^{\beta}\mathit{exp}$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
$c.m({}^{\beta_1}\mathit{arg}_1, \ldots, {}^{\beta_k}\mathit{arg}_k)$	$\mathit{before}(\beta_1) = \mathit{before}(\alpha)$ $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i)$, $i = \overline{1, k-1}$ $\mathit{after}(\alpha) = \mathit{after}(\beta_k) \cup$ $\cup\, \mathit{OutParams}(\mathit{arg}_1, \ldots, \mathit{arg}_k)$

Table 2: Definite assignment for arbitrary expressions

we simply state the equations they have to satisfy. Table 1 contains the equations for boolean expressions (see [Fru03b] for details). In addition, we have for all expressions in Table 1 the equation $\mathit{after}(\alpha) = \mathit{true}(\alpha) \cap \mathit{false}(\alpha)$. For a boolean expression α which is not an instance of one of the expressions in Table 1, we have $\mathit{true}(\alpha) = \mathit{after}(\alpha)$ and $\mathit{false}(\alpha) = \mathit{after}(\alpha)$.

Table 2 lists the equations specific to arbitrary expressions where *loc* stands for a local variable and *lit* for a literal. Note that following a method invocation, the `out` parameters $\mathit{OutParams}(\mathit{arg}_1, \ldots, \mathit{arg}_k)$ are definitely assigned. In cases not stated in Tables 1,2, if $^{\alpha}\mathit{exp}$ is an expression with *direct subexpressions* $^{\beta_1}e_1, \ldots, {}^{\beta_n}e_n$, then the left-to-right evaluation scheme yields the *general data flow equations*: $\mathit{before}(\beta_1) = \mathit{before}(\alpha)$, $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i)$, $i = \overline{1, n-1}$ and $\mathit{after}(\alpha) = \mathit{after}(\beta_n)$.

The equations specific to every statement can be found in Table 3. We assume that `try` statements are either `try-catch` or `try-finally` statements (see [Bor03] 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). This can be formalized as follows. For two statements α and β, we consider $\mathit{Fin}(\alpha, \beta)$ to be the list $[\gamma_1, \ldots, \gamma_n]$

$^{\alpha}\mathit{stm}$	the data flow equations
;	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
$({}^{\beta}\mathit{exp}$`;`$)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
$\{{}^{\beta_1}\mathit{stm}_1 \ldots {}^{\beta_n}\mathit{stm}_n\}$	$\mathit{before}(\beta_1) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta_n) \cap \mathit{vars}(\alpha)$ $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i) \cap$ $\cap\, \mathit{goto}(\beta_{i+1}), i = \overline{1, n-1}$
`if` $({}^{\beta}\mathit{exp})\ {}^{\gamma}\mathit{stm}_1$ `else` $^{\delta}\mathit{stm}_2$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{true}(\beta)$ $\mathit{before}(\delta) = \mathit{false}(\beta)$ $\mathit{after}(\alpha) = \mathit{after}(\gamma) \cap \mathit{after}(\delta)$
`while` $({}^{\beta}\mathit{exp})\ {}^{\gamma}\mathit{stm}$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{true}(\beta)$ $\mathit{after}(\alpha) = \mathit{false}(\beta) \cap \mathit{break}(\alpha)$
`goto L;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`break;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`continue;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`return;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`return` $^{\beta}\mathit{exp}$`;`	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`throw;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`throw` $^{\beta}\mathit{exp}$`;`	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`try` $^{\beta}\mathit{block}$ `catch`$(E_1\ x_1)\ {}^{\gamma_1}\mathit{block}_1$ $\vdots$ `catch`$(E_n\ x_n)\ {}^{\gamma_n}\mathit{block}_n$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma_i) = \mathit{before}(\alpha) \cup \{x_i\}$ $i = \overline{1, n}$ $\mathit{after}(\alpha) = \mathit{after}(\beta) \cap$ $\cap \bigcap_{i=1}^{n} \mathit{after}(\gamma_i)$
`try` $^{\beta}\mathit{block}_1$ `finally` $^{\gamma}\mathit{block}_2$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta) \cup$ $\cup\, \mathit{after}(\gamma)$

Table 3: Definite assignment for statements

of `finally` blocks of all `try-finally` statements in the innermost to outermost order from α to β. Then we define the set $\mathit{JoinFin}(\alpha, \beta)$ of definitely assigned variables after the execution of all these `finally` blocks: $\bigcup_{\gamma \in \mathit{Fin}(\alpha,\beta)} \mathit{after}(\gamma)$. Further, we define the set *goto* for a statement β. For a labeled statement $^{\beta}L : \mathit{stm}$, the set $\mathit{goto}(\beta)$ is given by $\bigcap_{^{\alpha}\texttt{goto L;}} (\mathit{before}(\alpha) \cup \mathit{JoinFin}(\alpha, \beta))$ where we take only the `goto` statements in the scope of β. For all the other statements, as well for a labeled statement with no `goto` statements, $\mathit{goto}(\beta)$ is the universal set $\mathit{vars}(\beta)$. Now we are able to state the equation $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i) \cap \mathit{goto}(\beta_{i+1})$ from Table 3. In case of a labeled statement, the equation formalizes the above stated idea while for a non-labeled statement becomes $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i)$.

The following example is a simplification of an example from the C# Specification [Wil03, §5.3.3.15]:

$\,$ `int i;`
$^{\delta}$`try {`$^{\alpha}$ `goto L;}`
`finally` $^{\gamma}$`{i = 3;}`
$^{\beta}$`L:Console.WriteLine(i);`

The C# Specification states that `i` is definitely assigned before β , i.e. $\texttt{i} \in$ *before*(β). Our equation *before*(β) = *after*$(\delta) \cap$ *goto*(β) led us to the same conclusion since *goto*(β) = *before*$(\alpha) \cup$ *after*(γ) and $\texttt{i} \in$ *after*$(\gamma) \subseteq$ *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: we get the error that `i` is unassigned. This problem was fixed in .NET Framework 1.1 but still exists in Rotor.

The following explanation holds for the equation *after*(α) = *after*$(\beta_n) \cap$ *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:

$\{^{\alpha}$`{int i;i = 1;} {int i;i = 2 *` $^{\beta}$`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 $\texttt{i} \notin$ *before*(β) and the block is rejected.

The idea for the equation which computes *after*(α) of a `while` statement α, is similar with that for a labeled statement. Similarly with the set *goto*, we define the set *break*(α) to be the set of variables definitely assigned before all corresponding `break` statements (and possibly after appropriate `finally` blocks). This means that the set *break*(α) is given by $\bigcap_{\beta \texttt{break;}}$(*before*$(\beta) \cup$ *JoinFin*(β,α)) 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 statements. Let α be the following statement:

`if(b)` $^{\gamma}$`{i = 1;} else` $^{\delta}$`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*(α) 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

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

then we have the following equations *after*$(\alpha) = \{\texttt{i}\}$, *before*$(\beta)$ = *after*$(\alpha) \cap$ *before*(γ) and *before*(γ) = *before*(β). After some simplification we find that *before*$(\beta) = \{\texttt{i}\} \cap$ *before*(β) and therefore we get two solutions for *before*(β) (and also for *before*(γ)): $\emptyset$ and $\{\texttt{i}\}$. 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 $\{\texttt{i}\}$ 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 $\{\texttt{i}\}$ and therefore the solution we require is the maximal fixed point *MFP*.

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 *Vars*(*meth*) of all local variables from the method body *mb*. We define the function $F : D \to D$ with $D = \mathcal{P}(\textit{Vars}(\textit{meth}))^r$ 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 *vars* are interpreted as constants.

We define now the relation $\sqsubseteq$ 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 $(X_1, \ldots, X_r) \sqsubseteq (X_1', \ldots, X_r')$ 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 and on the other hand, D is a lattice since it is a product of lattices: $(\mathcal{P}(\textit{Vars}(\textit{meth})), \subseteq)$ is a *poset* since the set inclusion is a partial order and for every two sets $X, Y \in \mathcal{P}(\textit{Vars}(\textit{meth}))$ there exists a lower bound $(X \cap Y)$ and an upper bound $(X \cup Y)$. $\square$

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. In order to prove the monotonicity of $F = (F_1, \dots, F_r)$, 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 $(\mathit{Vars}(\mathit{meth}), \dots, \mathit{Vars}(\mathit{meth}))$, 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 *MFP* corresponding to *before*(α), *after*(α), *true*(α) and *false*(α), respectively.

$^{\alpha}exp$	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

$^{\alpha}exp$	edges
loc	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
lit	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
$(\mathit{loc} = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$(\mathit{loc}\ \mathit{op} = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$(^{\beta}e_0$ `?` $^{\gamma}e_1$ `:` $^{\delta}e_2)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma))$ $(\mathcal{F}(\beta), \mathcal{B}(\delta)), (\mathcal{A}(\gamma), \mathcal{A}(\alpha)),$ $(\mathcal{A}(\delta), \mathcal{A}(\alpha))$
$c.f$	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
`ref` $^{\beta}exp$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
`out` $^{\beta}exp$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$c.m(^{\beta_1}\mathit{arg}_1, \dots, ^{\beta_k}\mathit{arg}_k)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), (\mathcal{A}(\beta_k), \mathcal{A}(\alpha)),$ $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), i = \overline{1, k-1}$

Table 5: Control flow for arbitrary expressions

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. 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 the graph are given by the *control transfer* defined in the C# Specification [Wil03, §8]. We show in Tables 4 and 5 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* $\beta_1, \dots, \beta_n$, then the left-to-right evaluation scheme adds to the flow graph also the following edges: $(\mathcal{B}(\alpha), \mathcal{B}(\beta_1))$,$(\mathcal{A}(\beta_n), \mathcal{A}(\alpha))$ and $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$, $i = \overline{1, n-1}$.

For each boolean expression α in Table 4, we have supplementary edges: $(\mathcal{T}(\alpha), \mathcal{A}(\alpha))$, $(\mathcal{F}(\alpha), \mathcal{A}(\alpha))$ 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 need to define also 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 to the graph the edges $(\mathcal{A}(\alpha), \mathcal{T}(\alpha))$, $(\mathcal{A}(\alpha), \mathcal{F}(\alpha))$. 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 $^{\beta}$`(!(`exp_1`|`exp_2`))`, then the 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 $\mathcal{A}(\alpha)$.

Table 6 introduces the edges of the control flow graph for each statement. Note that 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:

$^{\alpha}$`if` $^{\beta}$`(true||b)` $^{\delta}$`i = 1;`
`else` $^{\gamma}$`{int j = i;}`

Although the new considered test (i.e. `true`) cannot evaluate to `false`, we still add to the graph the edge $(\mathcal{F}(\beta),\mathcal{B}(\gamma))$ since anyway the false point of `true` is not reachable (see Table 4). In the presence of `finally` blocks, the jump statements `goto`, `break` and `continue` bring more complexity to the graph. Whenever such a jump statement exits one or more `try` blocks with associated `finally` blocks, the control is transferred first to the `finally` block (if any) of the innermost `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 `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 *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

$^{\alpha_1}$`try` $^{\alpha_2}$`{`
  $^{\alpha_3}$`(`$^{\alpha_4}$`(i = 1);)`
  `goto L;`
`} finally` $^{\alpha_5}$`{}`
$^{\alpha_6}$`L:Console.WriteLine(i);`

in the theoretical case when 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 $\mathcal{B}(\alpha)$ and the set $ThroughFin_a(\alpha,\beta)$ has 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 $ThroughFin_b(\alpha,\beta)$ has only the edge $(\mathcal{B}(\alpha),\mathcal{B}(\beta))$ while $ThroughFin_a(\alpha,\beta)$ refers to the edge $(\mathcal{B}(\alpha),\mathcal{A}(\beta))$.

$^{\alpha}stm$	edges
;	$(\mathcal{B}(\alpha),\mathcal{A}(\alpha))$
$(^{\beta}exp;)$	$(\mathcal{B}(\alpha),\mathcal{B}(\beta)),(\mathcal{A}(\beta),\mathcal{A}(\alpha))$
$\{^{\beta_1}stm_1\ldots{}^{\beta_n}stm_n\}$	$(\mathcal{B}(\alpha),\mathcal{B}(\beta_1)),(\mathcal{A}(\beta_n),\mathcal{A}(\alpha)),$ $(\mathcal{A}(\beta_i),\mathcal{B}(\beta_{i+1})),i=\overline{1,n-1}$
`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` wrt α
`continue;`	$ThroughFin_b(\alpha,\beta)$, where β is the nearest enclosing `while` wrt 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`$(E_1\ x_1)$ $^{\gamma_1}block_1$ $\vdots$ `catch`$(E_n\ x_n)$ $^{\gamma_n}block_n$	$(\mathcal{B}(\alpha),\mathcal{B}(\beta)),(\mathcal{A}(\beta),\mathcal{A}(\alpha))$ $(\mathcal{B}(\alpha),\mathcal{B}(\gamma_i)),(\mathcal{A}(\gamma_i),\mathcal{A}(\alpha)),$ $i=\overline{1,n}$
`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

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 the control is transferred to the entry point of the labeled statement and `while` statement, respectively, while in case of `break` the set $ThroughFin_a$ is considered, since at the end, the control is transferred to the end point of the `while` statement.

There are two more remarks concerning the `try` statement. Since in a `try` block can anytime occur 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 no `catch` clause 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 6).

The next remark is concerning 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 points 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 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 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` $^{\gamma}$`{}`

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. If α is an expression or a statement, then $path_b(\alpha)$ and $path_a(\alpha)$ 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 $\mathcal{A}(\alpha)$ of α, respectively. Moreover, if α is a boolean expression, then $path_t(\alpha)$ 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.

5 THE 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 equations in Section 2, the risk of accessing the value of an unassigned variable does not exist. The correctness means that, if the analysis infers a 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, i.e. on every path. A variable *loc* is assigned on a path if the path contains an *initialization* of *loc*: a simple assignment to *loc*, a method invocation for which *loc* is an `out` parameter or a `catch` clause whose exception variable is *loc*. We prove actually more than the correctness. We show that the components of the maximal fixed point are exactly (not only a *safe approximation* of) the sets of variables for which there is an *initialization* on every path to the appropriate point. To formalize this, we define the following sets. If α is an arbitrary expression or statement, then $\mathrm{AP}_b(\alpha)$ and $\mathrm{AP}_a(\alpha)$ denote the sets of variables in $vars(\alpha)$ (the variables in the scope of which α is) for which there exists an initialization on every path in $path_b(\alpha)$ and in $path_a(\alpha)$, respectively. For a boolean expression α, we have two more 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.

The following lemma is proved by induction over the abstract syntax tree, starting from the root of the method body. It claims that, the MFP sets of an expression or statement α, consist of variables in the scope of which α is (see [Fru03b] for details).

Lemma 4 *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 we have also $\mathrm{MFP}_t(\alpha) \subseteq vars(\alpha)$ and $\mathrm{MFP}_f(\alpha) \subseteq vars(\alpha)$.*

The correctness of the definite assignment analysis in C# is proved in the next 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 $\mathrm{MFP}_t(\alpha) \subseteq \mathrm{AP}_t(\alpha)$ and $\mathrm{MFP}_f(\alpha) \subseteq \mathrm{AP}_f(\alpha)$.*

Proof. We consider the following definitions. The set $\mathrm{AP}^n_b(\alpha)$ is defined in the same way as $\mathrm{AP}_b(\alpha)$, except that we consider only the paths of length less or equal than n. Similarly, we define also the sets $\mathrm{AP}^n_a(\alpha)$, $\mathrm{AP}^n_t(\alpha)$, $\mathrm{AP}^n_f(\alpha)$ (analogously, we have definitions for the sets of paths $path^n$). According to these definitions, the following set equalities hold for an arbitrary α: $\mathrm{AP}_b(\alpha) = \bigcap_n \mathrm{AP}^n_b(\alpha)$, $\mathrm{AP}_a(\alpha) = \bigcap_n \mathrm{AP}^n_a(\alpha)$ and if α is a boolean expression, then $\mathrm{AP}_t(\alpha) = \bigcap_n \mathrm{AP}^n_t(\alpha)$ and $\mathrm{AP}_f(\alpha) = \bigcap_n \mathrm{AP}^n_f(\alpha)$. Therefore to complete the proof, it suffices to show for every n: if α is an expression or statement, then $\mathrm{MFP}_b(\alpha) \subseteq \mathrm{AP}^n_b(\alpha)$ and $\mathrm{MFP}_a(\alpha) \subseteq \mathrm{AP}^n_a(\alpha)$ and in addition, if α is a boolean expression, $\mathrm{MFP}_t(\alpha) \subseteq \mathrm{AP}^n_t(\alpha)$ and $\mathrm{MFP}_f(\alpha) \subseteq \mathrm{AP}^n_f(\alpha)$. 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). There is no initialization of any local variable on this path and therefore we have $\mathrm{AP}^1_b(mb) = \emptyset$ which satisfies $\mathrm{MFP}_b(mb) \subseteq \mathrm{AP}^1_b(mb)$ since from the equations $\mathrm{MFP}_b(mb) = \emptyset$. From the definition of AP^1_a, we get $\mathrm{AP}^1_a(mb) = vars(mb) = \emptyset$ and from the equations of a block, we derive also $\mathrm{MFP}_a(mb) \subseteq vars(mb)$

and implicitly $\mathrm{MFP}_a(\mathit{mb}) \subseteq \mathrm{AP}^1_a(\mathit{mb})$. If $\alpha \neq \mathit{mb}$, then $\mathrm{AP}^1_b(\alpha) = \mathrm{AP}^1_a(\alpha) = \mathit{vars}(\alpha)$ and $\mathrm{AP}^1_t(\alpha) = \mathrm{AP}^1_f(\alpha) = \mathit{vars}(\alpha)$ (if α is a boolean expression) and the basis of induction is complete (see Lemma 4).

Induction step: we prove here only $\mathrm{MFP}_b(\beta_{i+1}) \subseteq \mathrm{AP}^{n+1}_b(\beta_{i+1})$ for a labeled statement β_{i+1} in a block (see Table 3). Let *loc* be a local variable in $\mathrm{MFP}_b(\beta_{i+1})$. If there are no `goto` statements pointing to β_{i+1}, then the proof is the same as for a `while` statement with no associated `continue` statements (see [Fru03b]). If there are `goto` statements which point to β_{i+1}, we prove that there exists an initialization of *loc* on every path to $\mathcal{B}(\beta_{i+1})$ of length at most $n+1$ that passes through a `goto` statement and possibly through `finally` blocks of enclosing `try` statements. Let p be such a path containing a `goto` statement α. The equations in Table 3 imply $\mathit{loc} \in \mathit{goto}(\beta_{i+1})$ and further $\mathit{loc} \in \mathrm{MFP}_b(\alpha) \cup \mathit{JoinFin}(\alpha, \beta_{i+1})$. If there are no `finally` blocks in $\mathit{Fin}(\alpha, \beta_{i+1})$, then $\mathit{JoinFin}(\alpha, \beta_{i+1}) = \emptyset$ and implicitly $\mathit{loc} \in \mathrm{MFP}_b(\alpha)$. Using the induction hypothesis, we obtain $\mathit{loc} \in \mathrm{AP}^n_b(\alpha)$ and therefore p should contain at least one initialization of *loc*. If $\mathit{Fin}(\alpha, \beta_{i+1})$ is non-empty, i.e. $\mathit{Fin} = [\gamma_1, \ldots, \gamma_k]$, then from the definition of the set $\mathit{JoinFin}(\alpha, \beta_{i+1})$, we get $\mathit{loc} \in \mathrm{MFP}_b(\alpha) \cup \bigcup_{j=1}^k \mathrm{MFP}_a(\gamma_j)$. The case $\mathit{loc} \in \mathrm{MFP}_b(\alpha)$ has been previously analyzed. If there is a `finally` block γ_j such that $\mathit{loc} \in \mathrm{MFP}_a(\gamma_j)$, then we get $\mathit{loc} \in \mathrm{AP}^n_a(\gamma_j)$ from the induction hypothesis. And since necessarily p contains $\mathcal{A}(\gamma_j)$, we are sure that p has one initialization of *loc*. Thus, we showed that each path to $\mathcal{B}(\beta_{i+1})$ of length at most $n+1$, contains an initialization of *loc*, i.e *loc* $\in \mathrm{AP}^{n+1}_b(\beta_{i+1})$. □

We can prove actually more: the *MFP* solution is not only an approximation of *AP* but it is perfect (Theorem 3). 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 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 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 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, for the data flow equations in Tables 1, 2, 3, the left-to-right subset relations where instead of the sets *before*, *after*, *true* and *false* we have the sets AP_b, AP_a AP_t and AP_f, respectively. For the complete proof we refer the reader to [Fru03b]. □

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.*

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 exists always 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#. 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 [Bor03]. During the attempts to build this model, there were discovered in [Fru03a] a few discrepancies between the C# Specification and different implementations of C#.

References

[Bor03] E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C#. Accepted for publication in journal Theoretical Computer Science, 2003

[Fru03a] N. G. Fruja. Specification and Implementation Problems for C#. In B. Thalheim and W. Zimmermann, editors, Abstract State Machines 2004, LNCS. Springer, 2004.

[Fru03b] N. G. Fruja. The correctness of the definite assignment analysis in C#. Technical Report, ETH Zürich. `http://www.inf.ethz.ch/~fruja`

[Gou02] J. Gough. Compiling for the .NET. Common Language Runtime (CLR). Prentice Hall, 2002.

[Gru00] D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen. Modern Compiler Design. Wiley, 2000

[Nie99] F. Nielson, H.R. Nielson, C. Hankin. Principles of Program Analysis. Springer–Verlag, 1999.

[Sta01] R. F. Stärk, J. Schmid, E. Börger. Java and the Java Virtual Machine–Definition, Verification, Validation. Springer–Verlag, 2001.

[Sch03] N. Schirmer. Java Definite Assignment in Isabelle/HOL. ECOOP Workshop on Formal Techniques for Java–like Programs, 2003.

[Wil03] S. Wiltamuth and A. Hejlsberg. C# Language Specification. MSDN, 2003