Abstract Interpretation-Based Certification of Assembly Code

Xavier Rival

École Normale Supérieure
45, rue d'Ulm, 75230, Paris cedex 5, France *
rival@di.ens.fr

Abstract. We present a method for analyzing assembly programs based on source program analysis and invariant translation. It is generic in the choice of an abstract domain for representing stores. This method is adapted to the design of certification tools for assembly programs generated by compiling programs written in an imperative language, without writing a specific compiler or modifying an existing one since invariant translation only uses standard debugging information. A prototype was developed for a procedural subset of the C language.

Keywords: Static program analysis; compilation; Abstract Interpretation.

1 Introduction

Critical software is concerned with safety and analyzing source programs may not be considered a sufficient guarantee. Indeed, compilers are complex pieces of software and may contain bugs. Therefore, there is a need for extending the certification to the assembly code itself, especially when dealing with highly critical software (as in aeronautics).

Moreover, the safety properties usually checked concern the actual execution of the program, that is, the assembly code. For instance, checking that a C program does not contain any out-of-bound array access is useful to know that the compiled program will not access a wrong part of the memory. Furthermore, the definition of the undesirable behaviors could also be architecture or compiler dependent, as is often the case for overflows. Indeed, the specification of languages like C often leaves these behaviors unspecified, for the sake of execution speed (this avoids handling what could be considered errors and may simplify the design of compilers). Therefore, certifying the assembly code provides much a better confidence in the code as it allows to make no more assumption on the semantics of the source language and on the correctness of the compiler.

Nevertheless, certifying assembly code is quite a hard task. It requires analyzing high-level properties (like state reachability) which is rather involved at

* This work was supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 program.

the assembly level, since part of the structure of the program is lost at compile time: the control structure is rather terse (branching to program points stored in registers), the data structure is difficult to reconstruct (various addressing modes like relative addressing). Proving the compiler formally and relying on the analysis of the source code would be a satisfactory solution but it would be too expensive since proving a compiler like in [2] is a huge amount of work and modifying the compiler forces to adapt the proof.

The solution proposed in this paper is to use the results of an analysis of the source code and the debugging information (information about the way the compilation is done, about the correspondence between source and assembly variables, program points) in order to reduce the task to handle at the assembly code level to the checking of a translated invariant. This process should not depend on the compiler itself but on the debugging information which is standard. We can imagine designing a certifying tool for a given language and a given architecture but generic in the compiler. This tool would prove the correctness of an assembly program P_a obtained by compiling a program P_s as follows: it would infer an invariant I_s for P_s, translate it to an invariant I_a, verify that I_a actually is an invariant for P_a and check that I_a entails correctness of P_a. The compiler itself is never proved which is a source of flexibility. The method presented here was formalized inside the Abstract Interpretation framework [5, 6], that provides an integrated view in a single framework of both static analysis [4, 3] and program transformations [7] (hence, compilation).

The implementation of a prototype gave encouraging results.

Related works: As an example of translation of invariants at compile time we can cite the Proof Carrying Code approach [13]: in this case the translation is handled by the compiler itself which has therefore to be adapted. Moreover the target language is also modified so as to be type-safe [12]. So, this approach is restricted to type-safe programming languages (excluding C, used in many critical systems). The VOC approach [18] generates proof obligations at compile time and then solves it, so as to prove each instance of compilation: the generated proof obligations entail the correctness of the transformation. This approach also requires the compiler to be instrumented. The method proposed in [14] is similar. Among direct analyses of assembly code we can cite some works that aim at determining low level execution properties like memory usage, cache behavior or worst case execution time [1, 8, 16, 17].

Section 2 formalizes compilation. Section 3 states the soundness of the invariant translation method. Section 4 details aspects of invariant checking. Section 5 presents implementation results ; Section 6 concludes.

2 Compilation as a program transformation

2.1 Abstract Interpretation and program transformations

Cousot and Cousot developed Abstract Interpretation [5, 6] as a way of deriving relationships between different semantics so as to provide approximate but computable answers to undecidable (or costly) problems. Note that approximations are always sound: if an abstract analyzer claims that a program satisfies a property, then it actually satisfies it.

Practically, the *concrete semantics* $[\![P]\!] \in D$ (for instance the *collecting semantics* of all the states of a transition system) of a program P provides the most precise description of the behavior of P. It can be expressed as the least fixpoint of a monotone semantic function F in a lattice D. Given a Galois connection $D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\sharp$, the *abstract semantics* of P is $[\![P]\!]^\sharp = \alpha([\![P]\!])$. Provided there exists a monotone abstract semantic function $F^\sharp$ such that $F^\sharp \circ \alpha = \alpha \circ F$ the abstract semantics can also be expressed as a least fixpoint $\mathrm{lfp}F^\sharp$ in the lattice $D^\sharp$ (thanks to the fixpoint transfer theorem of [15]). Most of the time the abstract semantics itself is not computable, so a computable and sound approximation of $[\![P]\!]^\sharp$ is derived by computing the least fixpoint of a function $F^\sharp$ such that $\alpha \circ F \sqsubseteq F^\sharp \circ \alpha$ and by using a widening operator [5] to enforce convergence.

Program transformations can also be handled in this framework [7]. A program transformation is a process that inputs a program P and outputs a program P' whose semantics can be expressed as a transformation of the semantics of P. A convenient semantics for defining the semantic transformation t_s (which may be an isomorphism) associated to the syntactic transformation t can be obtained by abstract interpretation of the standard semantics as shown below:

$$\begin{tikzcd}
P \arrow[r, mapsto, "\text{semantics}"] \arrow[d, "t"'] & {[\![P]\!]} \arrow[r, shift right, "\alpha'"'] & {[\![P]\!]^\sharp} \arrow[l, shift right, "\gamma"'] \arrow[d, "t_s"] \\
P' \arrow[r, mapsto, "\text{semantics}"'] & {[\![P']\!]} \arrow[r, shift right, "\alpha'"'] & {[\![P']\!]^\sharp} \arrow[l, shift right, "\gamma'"']
\end{tikzcd}$$

Formalizing compilation and some class of program analyses in this same framework will enable us to make them commute in some sense.

2.2 Source and assembly programs

A (source or assembly) program P is defined by the data of:

- its *store*: S_P is the set of the possible values for the store of P. We write R for the range of values for variables and V_P for the set of *store locations* (that is *variables* or *memory locations*) of P. In this setting, $S_P = V_P \to R$.
- its *control structure* (L_P, i_P, τ_P) where L_P is a set of *program points*, $i_P \in L_P$ is the *entry program point* and $\tau_P \subseteq (L_P \times S_P) \times (L_P \times S_P)$ is the *transition relation* of P: $((x, s), (y, t)) \in \tau_P$ if and only if, the execution of P after having reached program point x with store s, may continue at program point y, with store t. Non-determinism is allowed since τ_P is a relation.

Note that the notion of program point does not necessarily correspond to syntactic program points: a program point may be defined by a pair (l, s) where l is a syntactic point and a s is a stack in the case of procedural programs).

In the following, if $\mathcal{E}$ is a set, we note $\mathcal{E}^\star$ for the set of sequences of elements of $\mathcal{E}$ and $\mathbb{P}(\mathcal{E})$ for the powerset of $\mathcal{E}$. The concrete semantics $[\![P]\!]$ of a program P

is the set of the partial execution traces of P. It can be defined as a least fixpoint in the lattice $(\mathbb{P}((L_P \times S_P)^\star), \subseteq)$ by: $[\![P]\!] = \mathrm{lfp}_\emptyset^\subseteq F_P$ where $F_P : \mathbb{P}((L_P \times S_P)^\star) \to \mathbb{P}((L_P \times S_P)^\star)$ is the semantic function:

$$
\begin{aligned}
F_P(X) = \{&\langle (i_P, s)\rangle \mid s \in S_P\} \\
&\cup \{\langle (x_0, s_0), \ldots, (x_n, s_n), (x_{n+1}, s_{n+1})\rangle \mid \langle (x_0, s_0), \ldots, (x_n, s_n)\rangle \in X \\
&\quad \wedge ((x_n, s_n), (x_{n+1}, s_{n+1})) \in \tau_P\}
\end{aligned}
$$

The two following sections present a simple imperative source language and a simple assembly language that can be described in this setting.

2.3 A simple imperative language

The syntax of the simple source language $\mathcal{L}$ is shown in Fig. 1. Variables ($v \in \mathbb{V}$) are all supposed to be globals. Statements (`S`) are affectations, conditionals and loops. Blocks (`B`) are lists of statements. Expressions (`E`) all have integer type. Conditions (`C`) represent conditional expressions and have type boolean. A store (or *environment*) maps the variables of a program to integer values.

Since this is a model, overflows are not taken into account. An erroneous execution of a program is a trace that is stopped at a non-exit program point (typically because of a division by 0). An erroneous special state Ω is introduced for that purpose. The initial value of variables is not determined.

$$
\begin{array}{llll}
\mathtt{E} ::= n & (n \in \mathbb{Z}) & \mathtt{S} ::= v := \mathtt{E} & (v \in \mathbb{V}) \\
\quad \mid\ v & (v \in \mathbb{V}) & \quad \mid\ \mathbf{skip} \mid \mathbf{if}\ \mathtt{C}\ \mathbf{then}\ \mathtt{B}\ \mathbf{else}\ \mathtt{B} & \\
\quad \mid\ -\mathtt{E} \mid \mathtt{E} + \mathtt{E} \mid \mathtt{E} - \mathtt{E} \mid \mathtt{E} \star \mathtt{E} \mid \mathtt{E}/\mathtt{E} & & \quad \mid\ \mathbf{while}\ \mathtt{C}\ \mathbf{do}\ \mathtt{B} & \\
\mathtt{C} ::= \mathbf{true} \mid \mathbf{false} \mid \neg\mathtt{C} & & \mathtt{B} ::= \mathtt{S} \mid \mathtt{S};\ \mathtt{B} & \\
\quad \mid\ \mathtt{E} == \mathtt{E} \mid \mathtt{E} < \mathtt{E} \mid \mathtt{C} \wedge \mathtt{C} \mid \mathtt{C} \vee \mathtt{C} & & &
\end{array}
$$

Fig. 1. The simple language $\mathcal{L}$.

2.4 A simple assembly language

Since we restrict to the compilation of a simple language without arrays or procedures we consider a simplified assembly language $\mathcal{A}$, without relative addressing. The abstract machine provides the following store locations:

- registers: R_i $(0 \leq i \leq r - 1)$;
- memory cells (which are indexed by integers): $M[i]$ $(i \in \mathbb{N})$;
- a condition register CR: when a test is handled this flag is set. Branching may occur later, according to the value of the condition register (`LT` for less than, `EQ` for equal, `GT` for greater than).

An assembly program is defined by a set of instructions labeled by distinct integers. A label l is intuitively the value of the program counter when the instruction I_l is executed. The instructions and their semantics are detailed in Fig. 2. After the execution of a non branching instruction I_l the execution flows to I_{l+1}. As above an erroneous state Ω is introduced to handle blocking error case (division by 0).

Syntax	Instruction	Semantics (sketched)
`li` R, n	load integer	stores integer n in register R
`load` R, n	load from mem.	stores $M[n]$ in register R
`store` R, n	store in the memory	stores the value contained in register R in the memory location $M[n]$
`add` R_0, R_1, R_2 (`mul`, `div`...)	addition (and other arith. ops.)	adds the values contained in R_1 and R_2 and stores the result in register R_0
`cmp` R_0, R_1	comparison	reads n_0 in R_0, n_1 in R_1 and compares them: if $n_0 < n_1$, then CR is set to `LT` if $n_0 = n_1$, then CR is set to `EQ` if $n_0 > n_1$, then CR is set to `GT`
`b` l	branching	branches to label l
`bc`(C) l (C is $=, <$...)	condit. branch.	branches to l if the content of CR corresponds to condition C

Fig. 2. Assembly instructions

2.5 Compilation

The compilation of the source program P_s into the assembly program P_a is correct if the semantics of these two programs are somewhat tied: the execution of a statement of P_s should be simulated by the execution of one or several steps of P_a and conversely, a step of execution of P_a should lead to a state s, such that an assembly state s' related to a state of the source code P_s should be reachable in zero or several execution steps from s. This relation between source and assembly programs semantics is defined by relations between subsets of source and assembly program points and memory locations. Not all the program points of a compiled program correspond to a point in the source since one source statement might be compiled into a sequence of assembly statements. Similarly not all the store locations of the assembly program correspond to a store location of the source program: for instance a register may correspond to no variable. Reciprocally, a source program point may correspond to no assembly program point in case of dead-code elimination (and the same for store locations in case of variable elimination).

Most compilers provide debugging information that contain the mapping between subsets of source and assembly locations and program points.

Fig. 3 shows a very simple example of compilation without any optimization of a small piece of code. Variable x is associated to $M[0]$; point 1 of P_s is mapped to point 2 of P_a... The correctness of the compilation expresses that execution traces of P_s correspond to execution traces of P_a: if x has value v at point 1 for some run r of P_s, then there exists a "corresponding" run r' of P_a that reaches point 2, and such that at that point, $M[0]$ contains value v. Note that registers are excluded from this mapping: information about equalities between the content of assembly memory locations will be needed for the invariant checking step (in Sect. 4.2).

$\mathbb{V}_{P_s} = \{x\}$

0 :	$x := 0$;
1 :	**while** $x < 100$
2 :	{
3 :	$x := x + 1$;
4 :	}
5 :	exit

(a) source code P_s.

0 :	`li`	$R_0, 0$
1 :	`store`	$R_0, 0$
2 :	`load`	$R_0, 0$
3 :	`li`	$R_1, 100$
4 :	`cmp`	R_0, R_1
5 :	`bc(<)`	7
6 :	exit	
7 :	`load`	$R_0, 0$
8 :	`li`	$R_1, 1$
9 :	`add`	R_2, R_0, R_1
10 :	`store`	$R_2, 0$
11 :	`b`	2

(b) compiled code P_a.

source	asm. prog.
Store locations	
x	$M[0]$
Program points	
0	0
1	2
3	7
4	11
5	6

(c) translation information.

Fig. 3. An example of compilation.

The relation between the semantics of the source and the compiled program is built in two steps: we first restrict both semantics and then we assume a bijection between the restricted semantics. Let P_s be a source program and P_a an assembly program, defined by their sets of store locations V_s and V_a (as above we note $S_s = V_s \to R$ and $S_a = V_a \to R$ for the corresponding sets of stores) and their control structures (L_s, i_s, τ_s) and (L_a, i_a, τ_a). We first consider the case of the assembly program. Let $L_a^r \subseteq L_a$ and $V_a^r \subseteq V_a$ be subsets of the program points and of the store locations of P_a. We write S_a^r for the set of restricted stores $V_a^r \to R$. The store projection operator $\rho_a : S_a \to S_a^r$ is defined by $\forall s \in (V_a \to R),\ \rho_a(s) = s\ |_{V_a^r}$ where $s\ |_{V_a^r}$ denotes the restriction of the function s to V_a^r. The trace restriction operator Φ_a is defined as follows:

$$\Phi_a(\langle (x_0, s_0), \ldots, (x_n, s_n) \rangle) = \langle (x_{k_0}, \rho_a(s_{k_0})), \ldots, (x_{k_l}, \rho_a(s_{k_l})) \rangle$$

where $x_{k_0}, \ldots, x_{k_l}$ are exactly the program points belonging to L_a^r in the sequence $x_0, \ldots, x_n$ in the same order as they appear in $\langle (x_0, s_0), \ldots, (x_n, s_n) \rangle$.

Trace restriction defines an abstraction of the semantics of programs. We define the *restricted semantics* of P_a as the set of traces $[\![P_a]\!]_r = \alpha_a^r([\![P_a]\!])$ where $\alpha_a^r(\mathcal{E}) = \{\Phi_a(t) \mid t \in \mathcal{E}\}$. The function α_a^r defines a Galois connection:

$$(\mathbb{P}((L_a \times S_a)^\star), \subseteq) \underset{\alpha_a^r}{\overset{\gamma_a^r}{\leftrightarrows}} (\mathbb{P}((L_a^r \times S_a^r)^\star), \subseteq)\ .$$

In the same way, a restricted semantics can be defined for the source program P_s as an abstraction of the concrete trace semantics $[\![P_s]\!]$, by choosing $V_s^r \subseteq V_s$ and $L_s^r \subseteq L_s$. In most cases we do not wish to abstract away any variable of the source program and therefore $V_s^r = V_s$ (except in case of dead variable elimination). For generality, this abstraction α_s^r is defined as for the assembly code (we note as above $S_s^r = V_s^r \to R$):

$$[\![P_s]\!]_r = \alpha_s^r([\![P_s]\!]) \qquad \text{where} \qquad (\mathbb{P}((L_s \times S_s)^\star), \subseteq) \underset{\alpha_s^r}{\overset{\gamma_s^r}{\leftrightarrows}} (\mathbb{P}((L_s^r \times S_s^r)^\star), \subseteq) \ .$$

In the following, if f is a function $f : A \to B$, we note $\widehat{f}$ for the function $\mathbb{P}(A) \to \mathbb{P}(B)$, $[(\mathcal{E} \subseteq A) \mapsto \{f(x) \mid x \in \mathcal{E}\}]$.

The correctness of the compilation is defined as a correspondence between some source program points and some assembly program points (that is between L_s^r and L_a^r) and a correspondence between part of the store locations (that is between V_s^r and V_a^r). Generally, the relation between store locations depends on the program point. For the sake simplicity, we consider it does not.

Definition 1 (Correctness of compilation). *With the same notations as above, let $\pi_l : L_s^r \to L_a^r$ a bijection between source and assembly restricted program points and $\pi_s : S_s^r \to S_a^r$ a bijection between source and assembly restricted stores (usually given by a bijection $\pi_v : V_s^r \to V_a^r$ between store locations).*

Let π be the function defined by:

$$\begin{array}{rll} \pi : & (L_s^r \times S_s^r)^\star & \to (L_a^r \times S_a^r)^\star \\ & \langle (x_0, s_0), \ldots, (x_n, s_n) \rangle & \mapsto \langle (\pi_l(x_0), \pi_s(s_0)), \ldots, (\pi_l(x_n), \pi_s(s_n)) \rangle \ . \end{array}$$

Then the compilation $\mathfrak{c}$ *of P_s into P_a is* correct *with respect to the* translation information (π_l, π_s) *if and only if $\widehat{\pi}$ is a bijection between $[\![P_s]\!]_r$ and $[\![P_a]\!]_r$.*

$$\begin{tikzcd} P_s \arrow[r] \arrow[d, "\mathfrak{c}"'] & {[\![P_s]\!]} \arrow[r, shift right, "\alpha_s^r"'] & {[\![P_s]\!]_r} \arrow[l, shift right, "\gamma_s^r"'] \arrow[d, phantom, "\|\!\|\!\|\widehat{\pi}"] \\ P_a \arrow[r] & {[\![P_a]\!]} \arrow[r, shift right, "\alpha_a^r"'] & {[\![P_a]\!]_r} \arrow[l, shift right, "\gamma_a^r"'] \end{tikzcd} \ .$$

Remark 1 (Optimizations). As mentioned above, code or variable elimination based optimizations are handled by choosing π_s and π_l so as to get rid with the removed entities.

Many optimizations that change the structure can also be handled in this framework by defining program points in a non syntactic way. For instance in case of an unrolling of a loop L, a syntactic program point x of the source program in the loop L is mapped to two points in the assembly program: one for odd numbers of iterations and one for even numbers of iterations. Handling the optimization reduces to split x into two program points x_{odd} and x_{even}.

The formalization of compilation presented above is comparable to the transition systems of [18]. The advantage of formalizing compilation inside the Abstract Interpretation framework is to bring both static analysis and compilation (and possibly optimizations) into a single framework, which makes reasoning about the process more simple.

3 Analysis, compilation and invariant translation

3.1 Static program analysis and program transformation

This subsection introduces a class of static program analyses, practically large enough to answer many questions such as run-time errors detection. Roughly speaking a program analysis will be defined by an abstraction of the trace semantics of programs (in practice an over-approximation of the abstract semantics is computed). We also prove that the abstraction defining such a static analysis is orthogonal to the "restriction" abstraction done in the previous section.

Let us consider a program P whose store ranges in $S = V \to R$ and of control flow graph (L, i, τ). We keep the previous notations: we note L^r and V^r for the restricted sets of program points and variables, ρ for the store projection, α^r for the restriction abstraction. We suppose we are given an abstraction on the store, that is a Galois connection $(\mathbb{P}(S), \subseteq) \underset{\alpha^s}{\overset{\gamma^s}{\leftrightarrows}} (D^\sharp, \sqsubseteq)$.

The *abstract semantics* $[\![P]\!]^\sharp$ of the program P is obtained by partitioning $[\![P]\!]$ by the program points L and abstracting the sets of stores at each program point using α^s. Formally, this amounts to computing the abstraction α^t:

$$[\![P]\!]^\sharp = \alpha^t([\![P]\!]) \quad \text{where} \quad (\mathbb{P}((L \times S)^\star), \subseteq) \underset{\alpha^t}{\overset{\gamma^t}{\leftrightarrows}} (L \to D^\sharp, \dot{\sqsubseteq})$$
$$\text{and} \quad \alpha^t(\mathcal{E}) = [(x \in L) \mapsto \alpha^s(\{s \mid \langle \ldots, (x, s), \ldots \rangle \in \mathcal{E}\})] \ .$$

In some cases, the abstract semantics $[\![P]\!]^\sharp$ may also be computed directly as a least fixpoint of an abstract semantic function $F_P^\sharp$. However a static analyzer usually computes a sound approximation of $[\![P]\!]^\sharp$ by iterating a sound monotonic function $\overline{F}_P^\sharp$ and using widening to enforce convergence. Fig. 4(a) presents the result of a classical interval analysis [5] of the program of Fig. 3(a).

program point	x
0	$\top$
1	[0 ; 100]
3	[0 ; 99]
4	[1 ; 100]
5	[100 ; 100]

(a) Source analysis.

program point	$M[0]$
0	$\top$
2	[0 ; 100]
7	[0 ; 99]
11	[1 ; 100]
6	[100 ; 100]

(b) Translated invariant.

Fig. 4. Source analysis and invariant translation.

We now come to the second point of this subsection: the trace restriction abstraction used to define correctness of the compilation and the abstraction α^t corresponding to the program analysis are independent and can be commuted.

The first step to reach that goal is to design a *restricted abstract domain* $D^{r\sharp}$ for $S^r = V^r \to R$ and a projection $\rho^\sharp$ of $D^\sharp$ on $D^{r\sharp}$ such that the abstraction and the projection commute, that is a Galois connection

$$(\mathbb{P}(S^r), \subseteq) \xrightleftharpoons[\alpha^{sr}]{\gamma^{sr}} (D^{r\sharp}, \dot{\sqsubseteq}) \qquad \text{such that } \alpha^{sr} \circ \rho = \rho^\sharp \circ \alpha^s$$

This is in general easy. In the case of non relational domains, α^s is the pointwise abstraction of functions in $V \to R$ to functions in $V \to R^\sharp$. The same pointwise abstraction of functions in $V^r \to R$ to functions in $V^r \to R^\sharp$ commutes with domain restriction of functions. The case of most relational domains (like the octagons of [11] or the linear inequalities of [10]) is similar: forgetting the information about the variables of $V \setminus V^r$ is sufficient.

Then an abstraction on restricted traces can be defined as above by partitioning $[\![P]\!]_r$ by L^r and abstracting the sets of stores at each program point:

$$\begin{aligned} [\![P]\!]_r^\sharp = \alpha^{tc}([\![P]\!]_r) \quad & \text{where } (\mathbb{P}((L^r \times S^r)^\star), \subseteq) \xrightleftharpoons[\alpha^{tc}]{\gamma^{tc}} (L^r \to D^{r\sharp}, \dot{\sqsubseteq}) \\ & \text{and } \alpha^{tc}(\mathcal{E}) = [(x \in L^r) \mapsto \alpha^{sr}(\{s \mid \langle \dots, (x, s), \dots \rangle \in \mathcal{E}\})] \ . \end{aligned}$$

Moreover a program invariant $I \in (L \to D^\sharp)$ can be abstracted to an invariant $I^r = \alpha^{rc}(I) \in (L^r \to D^{r\sharp})$, α^{rc} being the abstract counterpart of α^r:

$$\begin{gathered} \forall x \in L^r,\ I^r(x) = \alpha^{rc}(I)(x) = \rho^\sharp(I(x)) \\ (L \to D^\sharp, \dot{\sqsubseteq}) \xrightleftharpoons[\alpha^{rc}]{\gamma^{rc}} (L^r \to D^{r\sharp}, \dot{\sqsubseteq}) \ . \end{gathered}$$

The relationship between the program analysis and the trace restriction used for formalizing the correctness of the compilation is stated by the theorem:

Theorem 1. *With the above notations (α^r denotes the restriction abstraction and α^t the program analysis abstraction), $\alpha^{tc} \circ \alpha^r = \alpha^{rc} \circ \alpha^t$. In other words, the* restricted semantics *$[\![P]\!]_r^\sharp$ satisfies: $[\![P]\!]_r^\sharp = \alpha^{tc} \circ \alpha^r([\![P]\!]) = \alpha^{rc} \circ \alpha^t([\![P]\!])$.*

In other words analyzing the program and then restricting the results of the analysis by forgetting the abstract store at some program points and the information about some store locations amounts to first restricting the sets of program points and of locations and then abstracting traces.

3.2 Invariant translation

We stated above the abstractions corresponding to the compilation and to the program analysis. We now define sound invariant translation procedures and show that they output sound invariants in presence of sound compilers and analyzers.

We instantiate the notations and results of Sect. 3.1 on a source program P_s and on an assembly program P_a. For $i \in \{s, a\}$,

- α_i^r is the restriction abstraction in the sense of Sect. 2.5,
- α_i^t is the abstraction corresponding to the static analysis as in Sect. 3.1,
- α_i^{rc} is the invariant restriction abstraction (introduced in Sect. 3.1),

- α_i^{tc} corresponds to static analysis from restricted semantics (Sect. 3.1).

Let (π_l, π_s) be translation information in the sense of Def. 1. An invariant translation procedure is a function $\pi_s^\sharp : D_s^{r\sharp} \to D_a^{r\sharp}$. It is sound if and only if it is the abstract counterpart of the concrete $\widehat{\pi}_s$:

Definition 2 (Sound invariant translation procedure). *The invariant translation function $\pi_s^\sharp$ is sound with respect to π_s if and only if:*

$$\forall S \subseteq S_a^r,\ \pi_s^\sharp \circ \alpha_s^{sr}(S) = \alpha_a^{sr} \circ \widehat{\pi}_s(S) \ .$$

For instance, in case of non relational domains the pointwise invariant translation (guided by the memory locations mapping π_v) is sound. Fig. 4(b) presents the translated invariant corresponding to the invariant of Fig. 4(a) (example of Fig. 3(a)).

Theorem 2 (Soundness of invariant translation). *If I_s is a sound abstract invariant for the source program P_s (i.e. $[\![P_s]\!]^\sharp \dot{\sqsubseteq} I_s$), if the compilation of P_s into P_a is correct with respect to (π_l, π_s) and if the invariant translation function $\pi_s^\sharp$ is correct, then the translated invariant $I_a = \pi_s^\sharp \circ \alpha_s^{rc}(I_s)$ is sound, that is: $[\![P_a]\!]_r^\sharp \dot{\sqsubseteq} I_a$.*

The proof of this result is done by composing the diagrams and applying straightforwardly the definitions, and twice Theorem 1. We first fix $I_s = [\![P_s]\!]^\sharp$:

$$\begin{tikzcd}
P_s \arrow[rrr, "\mathfrak{c}"] \arrow[d] & & & P_a \arrow[d] \\
{[\![P_s]\!]} \arrow[r, "\alpha_s^r"] \arrow[d, "\alpha_s^t"'] & {[\![P_s]\!]_r} \arrow[r, equal, "\pi_s"] \arrow[d, "\alpha_s^{tc}"'] & {[\![P_a]\!]_r} \arrow[d, "\alpha_a^{tc}"'] & {[\![P_a]\!]} \arrow[l, "\alpha_a^r"'] \arrow[d, "\alpha_a^t"'] \\
{[\![P_s]\!]^\sharp} \arrow[r, "\alpha_s^{rc}"] & {[\![P_s]\!]_r^\sharp} \arrow[r, "\pi_s^\sharp"] & {\pi_s^\sharp([\![P_s]\!]_r^\sharp) \dot{\sqsupseteq} [\![P_a]\!]_r^\sharp} & {[\![P_a]\!]^\sharp} \arrow[l, "\alpha_a^{rc}"']
\end{tikzcd} \ .$$

The general result of Theorem 2 follows: the translation functions and the abstraction functions are monotone ; soundness of I_s entails $[\![P_s]\!]^\sharp \dot{\sqsubseteq} I_s$.

The inequality $[\![P_a]\!]^\sharp \dot{\sqsubseteq} \gamma_a^{rc} \circ \pi_s^\sharp \circ \alpha_s^{rc}(I_s)$ is a direct consequence of the theorem (same hypotheses). Nevertheless the resulting approximation of $[\![P_a]\!]^\sharp$ is not precise enough, given $\forall x \in L_a \setminus L_a^r,\ \gamma_a^{rc} \circ \pi_s^\sharp \circ \alpha_s^{rc}(I_s) = \top$.

Sect. 4 addresses the problem of refining $I_a' = \gamma_a^{rc} \circ \pi_s^\sharp \circ \alpha_s^{rc}(I_s) = \gamma_a^{rc}(I_a)$ into an invariant I_a'' by invariant propagation and of checking that I_a'' is sound apart from any hypothesis about the correctness of the compiler or of the translator or even of the analyzer used for the source program.

4 Invariant checking

4.1 Invariant propagation and checking

We suppose here that an approximate abstract semantic function $\overline{F}_a^\sharp$ for the assembly program can be computed. Such a function defines an analyzer for the assembly program: iterating it starting from $\bot$ (using widening to enforce convergence) would lead to a sound invariant (which may be imprecise since direct analyses of assembly code are made difficult by the absence of a control structure adapted to efficient iteration). Anyway this function being monotone, it has a least fixpoint, which is also an approximation of $[\![P_a]\!]^\sharp$: $[\![P_a]\!]^\sharp \mathrel{\dot{\sqsubseteq}} \mathrm{lfp}\overline{F}^\sharp_a$.

Invariant checking. Checking that the translated invariant is sound reduces to verifying that I'_a is a post-fixpoint of $\overline{F}^\sharp_a : (L_a \to D^\sharp_a) \to (L_a \to D^\sharp_a)$. The choice of the abstract domain for assembly programs may be crucial (as in Sect. 4.2), to tackle the specificities of the assembly language and make sure $\overline{F}^\sharp_a$ can be defined so that I'_a indeed is a post-fixpoint. The checking could fail even if I'_a is sound, for instance if the verifier $\overline{F}^\sharp_a$ was too imprecise or if the assembly code contained some statement that would be very difficult to analyze precisely.

Invariant propagation. A common technique to refine a sound invariant is to iterate the semantic function starting from it: if it is a post-fixpoint then we get a decreasing sequence (which means we improve precision).

If the invariant I'_a computed in Sect. 3.2. is a post-fixpoint of $\overline{F}^\sharp_a$ then, the iterates of $\overline{F}^\sharp_a$ starting from I'_a form a decreasing sequence. Therefore computing a given number of iterates of this sequence leads to a more precise invariant.

Practical solution. The way of propagating the invariant and checking it we adopted is slightly different. The translated invariant I_a provides precise information for the points contained in L^r_a. In practice every branch of the assembly control flow graph contains at least one point x such that $x \in L^r_a$; in particular every cycle contains such a point. Therefore we define an element J_a of the abstract domain $L_a \to D^\sharp_a$ by

$$J_a : \begin{cases} x \in L^r_a \mapsto I_a(x) \\ x \notin L^r_a \mapsto \bot \end{cases},$$

and then we compute in one iteration a post-fixpoint of Φ starting from J_a, where Φ is defined by $\Phi(X) = X \mathrel{\dot{\cup}} \overline{F}^\sharp_a(X)$. In practice, we compute a local invariant for each node in the graph, by propagating local invariants forwards, using a work-list algorithm: the set of nodes a local invariant is known for (the so-called treated nodes) is initialized to L^r_a; then a local invariant can be computed for a node when a local invariant has already been determined for all its predecessors. When the process finishes a local invariant has been determined for any point in L_a since every cycle of the assembly control flow contains at least one point belonging to L^r_a. When all nodes got a local invariant, checking that the invariant is sound reduces to checking that for every node x, the local invariant of x is "implied" by the local invariants of the predecessors of x. This property should only be checked for the nodes of L^r_a since local invariants at the other nodes have been computed so as to achieve this property.

Theorem 2 shows that invariant translation yields a sound "restricted" invariant under some soundness hypotheses (that should be realized). This subsection showed how an invariant for the assembly program is reconstructed from the "restricted" one and how it is finally checked. Checking allows this invariant to be considered safe apart from any other hypothesis than the correctness of the checker, which is much a stronger guarantee. Indeed if the invariant checker is correct and claims the invariant is stable then the invariant is sound even if the compilation is not correct.

Note the checking may fail (for instance if some aspects of the assembly language are not analyzed precisely), which would not mean the restricted translated invariant would be incorrect.

4.2 Practical aspects of invariant propagation and checking

As mentioned above, invariant checking may require the use of a refined domain so as to handle the assembly language specificities. This section shows two of these together with their application to the example of Fig. 3.

Partitioning by the values of the CR : Conditional branching is commonly done in two steps in assembly languages (as in the language of Fig. 2): testing with modification of the condition register value according to the result of the comparison and branching according to the value of the condition register at branching time. Therefore the checker should propagate information about the condition register. In particular the local invariant at a point x should describe for any possible value $c \in \mathbb{C}$ of the condition register (where $\mathbb{C} = \{\mathtt{LT}, \mathtt{EQ}, \mathtt{GT}\}$) a precise over-approximation of the set S_c of stores that can be encountered at program point x and that map the condition register to the value c. With the notations of Sect. 3, this amounts to choosing $D_a^\sharp$ of the form $\mathbb{C} \to D'^\sharp_a$: an abstract value is a function that associates to each possible CR value v an abstract representation of a set of assembly stores whose CR is positioned to v. The abstract transition functions for testing and branching are given below:

- testing: we suppose a guard operator **guard** : $D_a^\sharp \times \mathtt{E} \to D_a^\sharp$ is provided. If P_a contains the instruction l : $\mathtt{cmp}\ R_O, R_1$, $I \in L_a \to D_a^\sharp$ and $\mathcal{I}$ is the contribution of the other predecessors of $l+1$:

$$\overline{F}_a^\sharp(I)(l+1) = \mathcal{I} \,\dot{\cup} \begin{cases} \mathtt{LT} \mapsto \mathbf{guard}(I(l), R_0 < R_1) \\ \mathtt{EQ} \mapsto \mathbf{guard}(I(l), R_0 = R_1) \\ \mathtt{GT} \mapsto \mathbf{guard}(I(l), R_0 > R_1) \end{cases}$$

- branching: we suppose that P_a contains the instruction l : $\mathtt{b}(\leq)l'$ and that $I \in L_a \to D_a^\sharp$. Then, if we define $\mathcal{I}$ and $\mathcal{I}'$ as above,

$$\overline{F}_a^\sharp(I)(l+1) = \mathcal{I} \,\dot{\cup} \begin{cases} \mathtt{LT} \mapsto \bot \\ \mathtt{EQ} \mapsto \bot \\ \mathtt{GT} \mapsto I(l)(\mathtt{GT}) \end{cases} \qquad \overline{F}_a^\sharp(I)(l') = \mathcal{I}' \,\dot{\cup} \begin{cases} \mathtt{LT} \mapsto I(l)(\mathtt{LT}) \\ \mathtt{EQ} \mapsto I(l)(\mathtt{EQ}) \\ \mathtt{GT} \mapsto \bot \end{cases} \quad .$$

Partitioning by the condition register value at each program point is not necessary (and would be prohibitively costly since common architectures provides several condition registers): information about the condition register (that is partitioning over condition register values) is only necessary "between" tests and branching nodes.

Equalities between assembly locations : A test on the value of a variable x stored in $M[i]$ is done in two steps: the value of the variable is copied into a register R_j and then the test is done on the register. Checking the invariant requires to take into consideration the fact that the value contained in $M[i]$ should be affected by the test. This can be done either by doing backwards iteration (which would be costly) or by using a domain precise enough to provide information of the form $a = b$ where a and b are memory locations. When implementing, we chose the last solution and implemented a domain whose abstract elements are the partitions of $\mathbb{P}(\mathbb{V})$ as in [9], where an element of a partition is a set of variables that store the same value for any execution at a given point.

Results : Fig. 5 displays the final stable invariant produced for the example of Fig. 3.

Beginning of line	Equalities	CR	R_0	R_1	R_2	$M[0]$
0	none	$\top$	$\top$	$\top$	$\top$	$\top$
1	none	$\top$	$[0;0]$	$\top$	$\top$	$\top$
2	none	$\top$	$[0;99]$	$\top$	$\top$	$[0;100]$
3	$R_0 = M[0]$	$\top$	$[0;100]$	$\top$	$\top$	$[0;100]$
4	$R_0 = M[0]$	$\top$	$[0;100]$	$[100;100]$	$\top$	$[0;100]$
5	$R_0 = M[0]$	LT	$[0;99]$	$[100;100]$	$\top$	$[0;99]$
5	$R_0 = M[0]$	EQ	$[100;100]$	$[100;100]$	$\top$	$[100;100]$
5	$R_0 = M[0]$	GT	$\bot$	$\bot$	$\bot$	$\bot$
6	$R_0 = M[0]$	$\top$	$[100;100]$	$[100;100]$	$\top$	$[100;100]$
7	$R_0 = M[0]$	$\top$	$[0;99]$	$[100;100]$	$\top$	$[0;99]$
8	$R_0 = M[0]$	$\top$	$[0;99]$	$[100;100]$	$\top$	$[0;99]$
9	$R_0 = M[0]$	$\top$	$[0;99]$	$[1;1]$	$\top$	$[0;99]$
10	$R_0 = M[0]$	$\top$	$[0;99]$	$[1;1]$	$[1;100]$	$[0;99]$
11	$R_2 = M[0]$	$\top$	$[0;99]$	$[1;1]$	$[1;100]$	$[1;100]$

Fig. 5. Reconstructed and checked invariant.

5 Implementation

A prototype was implemented for certifying Motorola PowerPC assembly code obtained by compiling C programs. Most features of the C language are handled (excluding pointers and recursion which should not be used in highly critical software), including functions, procedures, structures and arrays, standard integer and floating point data types (a restricted form of alias is permitted for arrays passed by reference to functions).

The analyzer is similar to the analyzer presented in [3]. The basic abstract domain is non relational (based on the domains of intervals for the floating point numbers and the integers and on the domain of constants for the booleans) but the expressiveness of the domain is notably improved by partitioning (by the values of variables as is the case of the condition register in assembly programs or by control paths-based criteria). At the assembly code level, various addressing modes are handled (absolute, relative) thanks to a symbolic representation of addresses and to the representation of the stack in the assembly abstract domain.

After an invariant has been proved to be sound at the assembly program level by the checker, the prototype attempts to certify the code by checking it cannot cause any of the following "runtime errors": division by 0, integer or floating point overflow, erroneous memory access (dereferencing of a wrong address). This prototype successfully certified assembly programs of thousands of instructions issued from the compilation of C programs of hundreds of lines including representative fragments of embedded systems. We can expect to certify much larger programs (the current version of the prototype stores one abstract store at each program point for the sake of programming simplicity and testing; this causes a huge memory requirement and is not necessary in a certifying tool, given propagation and safety checking could be done in one pass).

6 Conclusion and future work

We proposed a method for certifying assembly code produced by compilation from a language we have an analyzer for. The method is generic with respect to the compiler and to the choice of an abstract domain. Invariant propagation and checking may require a precise treatment of some assembly language aspects.

The approach proved to be successful in practice. Note that the final checking of the invariant is a strong guarantee: analyzing programs is a complex task, and checking at the end the result apart from any hypothesis on the correctness of the rest of the process is a good point. Moreover the distinct steps of the process are independent: the source analysis, the translation of the invariants and their checking can be done separately. Existing tools can be used which reduces the cost of the analysis of assembly programs.

A first extension of this work would be to turn the existing prototype into a true certifying tool, for instance by extending the abstract domain to relational domains. Another more challenging goal would be to define a class of transformations (optimizations...) the method would work for. A last direction would be to use similar methods to analyze programs generated automatically from a specification: the specification could be used to compute an invariant on the program; checking the invariant on the program being simpler than inferring an invariant from the generated program alone.

Acknowledgments. We would like to thank Bruno Blanchet, Patrick and Radhia Cousot, Jérôme Feret, Charles Hymans, Laurent Mauborgne, Antoine Miné, and David Monniaux for comments, suggestions and stimulating discussions.

References

1. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Static Analysis Symposium*, LNCS, 1996.
2. Y. Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, 1998.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D. Schmidt, and I. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS. Springer-Verlag, 2002. To appear.
4. P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Symposium on Principles of Programming Languages*, 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Symposium on Principles of Programming Languages.* ACM Press, New York, NY, 1979.
7. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the 29th Symposium on Principles of Programming Languages.* ACM Press, New York, NY, 2002.
8. C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS)*, 1997.
9. J. Feret. Dependency analysis of mobile systems. In *European Symposium on Programming (ESOP'02)*, 2002.
10. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.
11. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, 2001.
12. G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML Compiler: Performance and Safety Through Types. In *Workshop on Compiler Support for Systems Software*, 1996.
13. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, 1997.
14. G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.
15. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955.
16. H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.
17. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Seperate Cache and Path Analyses. *Real-Time Systems*, 2000.
18. L. Zuck, A. Pnuelli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. In J. Knoop and W. Zimmermann, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.