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 $D \xleftarrow{\gamma} D^2$, 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 $\text{fp} F^\sharp$ in the lattice $D^2$ (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 $\alpha \circ F \subseteq F^\sharp \circ \alpha$ 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 $t_x$ (which may be an isomorphism) associated to the syntactic transformation $t$ can be obtained by abstract interpretation of the standard semantics as shown below:

\[
P \xrightarrow{\text{semantics}} [P] \xleftarrow{\gamma} [P]^\sharp \xrightarrow{t_x} [P'] \xrightarrow{\gamma'} [P'^\sharp] \xrightarrow{\alpha'}
\]

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, $S_P = V_P \rightarrow R$.
- its control structure $(L_P, i_P, \tau_P)$ 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 $\tau_P$ 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 $s$ is a stack in the case of procedural programs).

In the following, if $E$ is a set, we note $E^*$ for the set of sequences of elements of $E$ and $P(E)$ for the powerset of $E$. 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 $\mathcal{P}(\mathcal{L}_P \times \mathcal{S}_P)^*$ by: $\mathcal{P} = \text{inf}_{\mathcal{P}} \mathcal{F}_P$ where $F_P : \mathcal{P}(\mathcal{L}_P \times \mathcal{S}_P)^* \rightarrow \mathcal{P}(\mathcal{L}_P \times \mathcal{S}_P)^*$ is the semantic function:

$$F_P(X) = \left\{ (i, s) \mid s \in \mathcal{S}_P \right\} \cup \left\{ ((x_0, s_0), \ldots, (x_n, s_n), (x_{n+1}, s_{n+1})) \mid (x_0, s_0), \ldots, (x_n, s_n) \in X \wedge ((x_n, s_n), (x_{n+1}, s_{n+1})) \in \gamma_P \right\}$$

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 ($v \in \mathcal{V}$) are all supposed to be globals. Statements ($\mathcal{S}$) are affectations, conditionals and loops. Blocks ($\mathcal{B}$) are lists of statements. Expressions ($\mathcal{E}$) all have integer type. Conditions ($\mathcal{C}$) 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 $\Omega$ is introduced for that purpose. The initial value of variables is not determined.

$$\mathcal{E} ::= n \quad (n \in \mathcal{Z}) \quad \mathcal{S} ::= v := \mathcal{E} \quad (v \in \mathcal{V})$$

$$\quad | \quad \text{skip} \quad | \quad \text{if } c \text{ then } \mathcal{B} \text{ else } \mathcal{B}$$

$$\quad | \quad \text{while } c \text{ do } \mathcal{B}$$

$$\mathcal{C} ::= \text{true} | \text{false} | -c \quad \mathcal{B} ::= \mathcal{S} | \mathcal{S}; \mathcal{B}$$

$$\quad | \quad \mathcal{E} = \mathcal{E} \quad | \quad \mathcal{E} < \mathcal{E} \quad | \quad \mathcal{C} \land \mathcal{C} \quad | \quad \mathcal{C} \lor \mathcal{C}$$

**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): $M[i]$ ($i \in \mathcal{N}$);
- 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 ($\text{LT}$ for less than, $\text{EQ}$ for equal, $\text{GT}$ 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-branched instruction $I_l$, the execution flows to $I_{l+1}$. As above, an erroneous state $\Omega$ is introduced to handle blocking error case (division by 0).
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Instruction</th>
<th>Semantics (sketched)</th>
</tr>
</thead>
<tbody>
<tr>
<td>li R, n</td>
<td>load integer</td>
<td>stores integer n in register R</td>
</tr>
<tr>
<td>load R, n</td>
<td>load from mem.</td>
<td>stores ( M[n] ) in register R</td>
</tr>
<tr>
<td>store R, n</td>
<td>store in the memory</td>
<td>stores the value contained in register R in the memory location ( M[n] )</td>
</tr>
<tr>
<td>add R0, R1, R2</td>
<td>addition (and other arith. ops.)</td>
<td>adds the values contained in R1 and R2 and stores the result in register R0</td>
</tr>
<tr>
<td>cmp R0, R1</td>
<td>comparison</td>
<td>reads ( n_0 ) in R0, ( n_1 ) in R1 and compares them: ( p = \begin{cases} \text{LT} &amp; \text{if } n_0 &lt; n_1, \ \text{EQ} &amp; \text{if } n_0 = n_1, \ \text{GT} &amp; \text{if } n_0 &gt; n_1. \end{cases} )</td>
</tr>
<tr>
<td>b l</td>
<td>branching</td>
<td>branches to label l</td>
</tr>
<tr>
<td>bcc(C) l</td>
<td>condit. branch.</td>
<td>branches to l if the content of CR corresponds to condition C</td>
</tr>
</tbody>
</table>

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 \( s' \) 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).
\[ V_{P_s} = \{ x \} \]

0: \( x := 0 \);
1: while \( x < 100 \)
2: \( x := x + 1 \);
3: exit
4: \( \text{load} \ R_0, 0 \)
5: \( \text{add} \ R_2, R_0, R_1 \)
6: \( \text{store} \ R_2, 0 \)
7: \( \text{bc}(<) \) 7
8: \( \text{li} \ R_1, 1 \)
9: \( \text{li} \ R_0, 0 \)
10: \( \text{add} \ R_2, R_0, R_1 \)
11: \( \text{b} \) 2

(b) compiled code \( P_c \).

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 \( S_s = V_s \rightarrow R \) and \( S_a = V_a \rightarrow R \) for the corresponding sets of stores) and their control structures \( (L_s, i_s, \tau_s) \) and \( (L_a, i_a, \tau_a) \). We first consider the case of the assembly program. Let \( L_s^r \subseteq L_s \) and \( V_s^r \subseteq V_s \) be subsets of the program points and of the store locations of \( P_a \). We write \( S_s^r \) for the set of restricted stores \( V_s^r \rightarrow R \). The store projection operator \( \rho : S_a \rightarrow S_s^r \) is defined by \( \forall s \in (V_a \rightarrow R), \rho(s) = s \mid_{V_s^r} \) where \( s \mid_{V_s^r} \) denotes the restriction of the function \( s \) to \( V_s^r \). The trace restriction operator \( \Phi \) is defined as follows:

\[
\Phi_a((x_0, s_0), \ldots, (x_n, s_n)) = (\langle x_{k_0}, \rho_a(s_{k_0}) \rangle, \ldots, \langle x_{k_i}, \rho_a(s_{k_i}) \rangle)
\]

where \( x_{k_0}, \ldots, x_{k_i} \) are exactly the program points belonging to \( L_s^r \) 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 \( [P_a]_r = \alpha_a^r([P_a]) \) where \( \alpha_a^r(\mathcal{E}) = \{ \Phi_a(t) \mid t \in \mathcal{E} \} \). The function \( \alpha_a^r \) defines a Galois connection:

\[
(\mathcal{P}((L_a \times S_a)^*), \subseteq) \xleftarrow{\gamma_a^r} (\mathcal{P}((L_a^r \times S_s^r)^*), \subseteq) \xrightarrow{\alpha_a^r} \mathcal{P}((L_a^r \times S_s^r)^*), \subseteq).
\]

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 \( \alpha_s^r \) is defined as for the assembly.
code (we note as above $S'_s = V'_s \rightarrow R$):

$$\left[ P_s \right]_r = \alpha'_s(\left[ P_s \right]) \quad \text{where} \quad (\mathbb{P}(\mathbb{I}_s \times S'_s), \subseteq) \xleftarrow{\gamma'_s} (\mathbb{P}(\mathbb{I}'_s \times S'_s), \subseteq).$$

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

The correctness of the compilation is defined as a correspondence between some source program points and some assembly program points (that is between $L'_s$ and $L'_a$) and a correspondence between part of the store locations (that is between $V'_s$ and $V'_a$). 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 $\pi_s : L'_s \rightarrow L'_a$ a bijection between source and assembly restricted program points and $\pi_s : S'_s \rightarrow S'_a$ a bijection between source and assembly restricted stores (usually given by a bijection $\pi_s : V'_s \rightarrow V'_a$ between store locations).

Let $\pi$ be the function defined by:

$$\pi : \quad (L'_s \times S'_s)^* \rightarrow (L'_a \times S'_a)^* \quad \langle(x_0, s_0), \ldots, (x_n, s_n)\rangle \mapsto \langle(\pi_s(x_0), \pi_s(s_0)), \ldots, (\pi_s(x_n), \pi_s(s_n))\rangle.$$

Then the compilation $c$ of $P_s$ into $P_a$ is correct with respect to the translation information $(\pi_i, \pi_a)$ if and only if $\tilde{\pi}$ is a bijection between $[P_s]_r$ and $[P_a]_r$.

$$\begin{align*}
P_s & \quad \xrightarrow{\pi} \quad [P_s] \quad \xleftarrow{\gamma'_s} \quad [P_s]_r \\
\downarrow & \quad \| \tilde{\pi} \| \\
P_a & \quad \xrightarrow{\pi} \quad [P_a] \quad \xleftarrow{\gamma'_a} \quad [P_a]_r.
\end{align*}$$

**Remark 1 (Optimizations).** As mentioned above, code or variable elimination based optimizations are handled by choosing $\pi_s$ and $\pi_i$ 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 $S = V \rightarrow R$ and of control flow graph $(L, i, \tau)$. We keep the previous notations: we note $L'$ and $V'$ for the restricted sets of program points and variables, $\rho$ for the store projection, $\alpha^r$ for the restriction abstraction. We suppose we are given an abstraction on the store, that is a Galois connection $(\mathcal{P}(S), \subseteq) \xleftarrow{\gamma^r} (D^g, \subseteq)$.

The abstract semantics $[P]^g$ 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 $\alpha^r$. Formally, this amounts to computing the abstraction $\alpha^i$:

$$[P]^g = \alpha^i([P]) \quad \text{where} \quad (\mathcal{P}(L \times S^r), \subseteq) \xleftarrow{\gamma^i} (L \rightarrow D^g, \subseteq)$$

and

$$\begin{align*}
\alpha^i(E) = [x \in L \mapsto \alpha^r([s \mid (\ldots, (x,s), \ldots) \in E])]
\end{align*}$$

In some cases, the abstract semantics $[P]^g$ may also be computed directly as a least fixpoint of an abstract semantic function $F^g_P$. However a static analyzer usually computes a sound approximation of $[P]^g$ by iterating a sound monotonic function $F^g_P$ 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).

<table>
<thead>
<tr>
<th>program point</th>
<th>$x$</th>
<th>program point</th>
<th>$M[0]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\top$</td>
<td>0</td>
<td>$\top$</td>
</tr>
<tr>
<td>1</td>
<td>[0; 100]</td>
<td>2</td>
<td>[0; 100]</td>
</tr>
<tr>
<td>3</td>
<td>[0; 99]</td>
<td>7</td>
<td>[0; 99]</td>
</tr>
<tr>
<td>4</td>
<td>[1; 100]</td>
<td>11</td>
<td>[1; 100]</td>
</tr>
<tr>
<td>5</td>
<td>[100; 100]</td>
<td>6</td>
<td>[100; 100]</td>
</tr>
</tbody>
</table>

(a) Source analysis. (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 $\alpha^i$ 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 $D^{r\pi}$ for $S^r = V' \rightarrow R$ and a projection $\rho^g$ of $D^g$ on $D^{r\pi}$ such that the abstraction
and the projection commute, that is a Galois connection

$$ (\mathcal{P}(S^r), \subseteq) \xrightarrow{\gamma^r} (D^{r\emptyset}, \sqsubseteq) $$

such that $$ \alpha^{sr} \circ \rho = \rho^d \circ \alpha^s $$

This is in general easy. In the case of non relational domains, $$ \alpha^s $$ is the pointwise abstraction of functions in $$ V \rightarrow R $$ to functions in $$ V \rightarrow R^d $$. The same pointwise abstraction of functions in $$ V^r \rightarrow R $$ to functions in $$ V^r \rightarrow R^d $$ 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 $$ [P]_e $$ by $$ L^r $$ and abstracting the sets of stores at each program point:

$$ [P]_r^{\alpha} = \alpha^{rc}([P]_r) $$

where $$ (\mathcal{P}((L^r \times S^r)^*), \subseteq) \xrightarrow{\gamma^{rc}} (L^r \rightarrow D^{r\emptyset}, \sqsubseteq) $$

and $$ \alpha^{rc}(\mathcal{E}) = [(x \in L^r) \mapsto \alpha^r([s \mid (\ldots, (x, s), \ldots) \in \mathcal{E}])] $$ .

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

$$ \forall x \in L^r, I^r(x) = \alpha^{rc}(I)(x) = \rho^d(I(x)) $$

$$ (L \rightarrow D^\emptyset, \subseteq) \xrightarrow{\gamma^{rc}} (L^r \rightarrow D^{r\emptyset}, \sqsubseteq) $$ .

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 ($$ \alpha^r $$ denotes the restriction abstraction and $$ \alpha^s $$ the program analysis abstraction), $$ \alpha^{tc} \circ \alpha^r = \alpha^r \circ \alpha^{t} $$ In other words, the restricted semantics $$ [P]_r^{\alpha} $$ satisfies:

$$ [P]_r^{\alpha} = \alpha^{tc} \circ \alpha^r([P]) = \alpha^r \circ \alpha^{t}([P]). $$

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 $$ i \in \{s, a\} $$,

- $$ \alpha^r_i $$ is the restriction abstraction in the sense of Sect. 2.5,
- $$ \alpha^s_i $$ is the abstraction corresponding to the static analysis as in Sect. 3.1,
- $$ \alpha^{rc}_i $$ is the invariant restriction abstraction (introduced in Sect. 3.1),
- $\alpha^{lc}_i$ corresponds to static analysis from restricted semantics (Sect. 3.1).

Let $(\pi_i, \pi_a)$ be translation information in the sense of Def. 1. An invariant translation procedure is a function $\pi^{a}_s : D^{\ast}_s \rightarrow D^{\ast}_a$. It is sound if and only if it is the abstract counterpart of the concrete $\pi_s$:

**Definition 2 (Sound invariant translation procedure).** The invariant translation function $\pi^{a}_s$ is sound with respect to $\pi_a$ if and only if:

$$\forall S \subseteq S^{a}_a, \quad \pi^{a}_s(\pi^{a}_s(S)) = \pi^{a}_a \circ \pi^{a}_s(S).$$

For instance, in case of non-relational domains the pointwise invariant translation (guided by the memory locations mapping $\pi_s$) 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 $I_a$ is a sound abstract invariant for the source program $P_s$ (i.e. $[P_s]^{a} \subseteq I_a$), if the compilation of $P_s$ into $P_a$ is correct with respect to $(\pi_i, \pi_a)$ and if the invariant translation function $\pi^{a}_s$ is correct, then the translated invariant $I_a = \pi^{a}_s \circ \pi^{a}_s(I_a)$ is sound, that is: $[P_a]^{a} \subseteq I_a$.

The proof of this result is done by composing the diagrams and applying straightforwardly the definitions, and twice Theorem 1. We first fix $I_a = [P_s]^{a}$:

\[
\begin{array}{ccc}
P_s & \overset{c}{\longrightarrow} & P_a \\
\downarrow & & \downarrow \\
[P_s] & \overset{\alpha^{s}_s}{\longrightarrow} & [P_a], \\
\downarrow & & \downarrow \\
[P_s]^{a} & \overset{\pi^{a}_a \circ \pi^{a}_s}{\longrightarrow} & [P_a]^{a}, \\
\downarrow & & \downarrow \\
[P_a]^{a} & \overset{\alpha^{a}_a \circ \pi^{a}_s}{\longrightarrow} & [P_a]^{a}. \\
\end{array}
\]

The general result of Theorem 2 follows: the translation functions and the abstraction functions are monotone; soundness of $I_a$ entails $[P_s]^{a} \subseteq I_a$.

The inequality $[P_a]^{a} \subseteq \gamma^{ac}_a \circ [P_s]^{a} \circ \alpha^{a}_s(I_a)$ is a direct consequence of the theorem (same hypotheses). Nevertheless the resulting approximation of $[P_a]^{a}$ is not precise enough, given $\forall x \in I_a \setminus I_a^{a}$, $\gamma^{ac}_a \circ [P_s]^{a} \circ \alpha^{a}_s(I_a) \supseteq I_a^{a}$.

Sect. 4 addresses the problem of refining $I_a^{a} = \gamma^{ac}_a \circ [P_s]^{a} \circ \alpha^{a}_s(I_a)$ into an invariant $I_a^{a}$ by invariant propagation and of checking that $I_a^{a}$ 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 abstract function $\overline{F}_a$ 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 \([P_a]^2 \subseteq \text{fp} \Phi_a\).

**Invariant checking.** Checking that the translated invariant is sound reduces to verifying that \( I_a' \) is a post-fixpoint of \( \Phi_a' : (L_a \rightarrow D_a^p) \rightarrow (L_a \rightarrow D_a^p) \). 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 \( \Phi_a^d \) 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 \( \Phi_a' \) 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 \( \Phi_a' \), then, the iterates of \( \Phi_a^d \) 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 \( L_a' \). In practice every branch of the assembly control flow graph contains at least one point \( x \) such that \( x \in L_a' \); in particular every cycle contains such a point. Therefore we define an element \( J_a \) of the abstract domain \( L_a \rightarrow D_a^p \) by

\[
J_a : \begin{cases} 
  x \in L_a' \mapsto I_a(x) \\
  x \not\in L_a' \mapsto \bot
\end{cases}
\]

and then we compute in one iteration a post-fixpoint of \( \Phi \) starting from \( J_a \), where \( \Phi \) is defined by \( \Phi(X) = X \cup \Phi_a^d(X) \). 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 \( L_a' \); 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 \( L_a' \). 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 \( L_a' \) 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 constructed 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 \( c \in C \) of the condition register (where \( C = \{ LT, EQ, GT \} \)) 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 \( D^a_{\alpha} \) of the form \( C \rightarrow D^a_{\alpha} \): 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 \( \text{guard} : D^a_{\alpha} \times E \rightarrow D^a_{\alpha} \) is provided.
  If \( P_a \) contains the instruction \( l : \text{cmp} \ R_0, R_1, I \in L_a \rightarrow D^a_{\alpha} \) and \( I \) is the contribution of the other predecessors of \( l + 1 \):
  \[
  T^a_{\alpha}(I)(l + 1) = I \cup \begin{cases} 
  LT \mapsto \text{guard}(I(l), R_0 < R_1) \\
  EQ \mapsto \text{guard}(I(l), R_0 = R_1) \\
  GT \mapsto \text{guard}(I(l), R_0 > R_1)
  \end{cases}
  \]

- branching: we suppose that \( P_a \) contains the instruction \( l : b(\leq) \) and that \( I \in L_a \rightarrow D^a_{\alpha} \). Then, if we define \( I \) and \( I' \) as above,
  \[
  T^a_{\alpha}(I)(l + 1) = I \cup \begin{cases} 
  LT \mapsto \bot \\
  EQ \mapsto \bot \\
  GT \mapsto I(l)(GT)
  \end{cases} \quad T^a_{\alpha}(I)(l') = I' \cup \begin{cases} 
  LT \mapsto I(l)(LT) \\
  EQ \mapsto I(l)(EQ) \\
  GT \mapsto \bot
  \end{cases}
  \]
Partitioning by the condition register value at each program point is not necessary (and would be prohibitively costly since common architectures provide 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 $P(V)$ 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.

<table>
<thead>
<tr>
<th>Beginning of line</th>
<th>Equalities</th>
<th>$CR$</th>
<th>$R_0$</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$M[0]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>none</td>
<td>T</td>
<td>[0;0]</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>none</td>
<td>T</td>
<td>[0;99]</td>
<td>T</td>
<td>T</td>
<td>[0;100]</td>
</tr>
<tr>
<td>3</td>
<td>$R_0 = M[0]$</td>
<td>T</td>
<td>[0;100]</td>
<td>T</td>
<td>T</td>
<td>[0;100]</td>
</tr>
<tr>
<td>4</td>
<td>$R_0 = M[0]$</td>
<td>T</td>
<td>[100;100]</td>
<td>T</td>
<td>T</td>
<td>[100;100]</td>
</tr>
<tr>
<td>5</td>
<td>$R_0 = M[0]$</td>
<td>LT</td>
<td>[0;99]</td>
<td>[100;100]</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>$R_0 = M[0]$</td>
<td>EQ</td>
<td>[100;100]</td>
<td>[100;100]</td>
<td>T</td>
<td>[100;100]</td>
</tr>
<tr>
<td>6</td>
<td>$R_0 = M[0]$</td>
<td>CT</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>7</td>
<td>$R_0 = M[0]$</td>
<td>T</td>
<td>[100;100]</td>
<td>[100;100]</td>
<td>T</td>
<td>[100;100]</td>
</tr>
<tr>
<td>8</td>
<td>$R_0 = M[0]$</td>
<td>T</td>
<td>[0;99]</td>
<td>[100;100]</td>
<td>T</td>
<td>[0;99]</td>
</tr>
<tr>
<td>9</td>
<td>$R_0 = M[0]$</td>
<td>T</td>
<td>[0;99]</td>
<td>[1;1]</td>
<td>T</td>
<td>[0;99]</td>
</tr>
<tr>
<td>10</td>
<td>$R_0 = M[0]$</td>
<td>T</td>
<td>[0;99]</td>
<td>[1;1]</td>
<td>[1;100]</td>
<td>[0;99]</td>
</tr>
<tr>
<td>11</td>
<td>$R_0 = M[0]$</td>
<td>T</td>
<td>[0;99]</td>
<td>[1;1]</td>
<td>[1;100]</td>
<td>[1;100]</td>
</tr>
</tbody>
</table>

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 ad-
dress). This prototype successfully certified assembly programs of thousands of
instructions issued from the compilation of C programs of hundreds of lines in-
cluding 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 transfor-
mations (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, Jéréôme Feret, Charles Hymans, Laurent Mauborgne, Antoine Miné, and
David Monniaux for comments, suggestions and stimulating discussions.
References